Preserving Architectural Decisions through Architectural Patterns

Minh Tu Ton That

THÈSE / UNIVERSITÉ DE BRETAGNE SUD

UFR Sciences et Sciences de l'Ingénieur

sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD

Mention : STIC

École Doctorale SICMA

présentée par

Minh Tu TON THAT

IRISA

Preserving Architectural Decisions through Architectural Patterns

Thèse soutenue le 30 Octobre 2014,
devant la commission d'examen composée de :

M. Jean-Marc Jézéquel
Professor, IRISA, Université de Rennes 1 / President

M. Danny Weyns
Professor, Linnaeus University / Reviewer

M. Khalil Drira
Professor, LASS, Toulouse / Reviewer

M. Jean-Marc Jézéquel
Professor, IRISA, Université de Rennes 1 / Examiner

M. Flavio Oquendo
Professor, IRISA, Université de Bretagne Sud, France / Advisor

M. Salah Sadou
Assistant Professor, IRISA, Université de Bretagne Sud, France / Advisor

Acknowledgements

I thank my supervisor Assoc. Prof. Dr. Salah Sadou for an exceptional supervision and guidance of my Ph.D. project. Thank you for all the patience and care you gave me since the first day I came to Vannes.

I thank my supervisor Prof. Dr. Flavio Oquendo for you kindness and inspiration. Thank you for your encouragement and supportive discussions despite busy schedules. Thank you for all the support during these three years.

I thank my colleagues at the ArchWare team at the University of South Brittany for numerous helpful discussions. I also thank all the people I met and had discussions with during conferences and workshops.

It has been a pleasure to be part of IRISA-UBS the last three years. I enjoyed plenty of talks in a variety of subjects during lunch time and hangouts with other PhD students. In particular, I loved to get to know you, Salma and Abdel. You have been such good friends, always caring and supportive.

Last but not least, I thank my darling Khanh Ha and my family for supporting me during this long journey. I thank you for your unconditional love and sacrifice you made for me. I would have not been able to go this far without you.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Motivation

Today's software systems tend to evolve over time to adapt to changes, which are supposed to be inevitable in software development [9]. No matter how well we design an application, there will always be changes about functionalities, performance, deployment requirements,... to name a few. The maintenance cost therefore sometimes outweighs the development cost. This phenomenon in modern software development shifts the focus from creating software to extending and adapting software. In other words, system designers on one hand, have to fulfil their actual task and on other hand, have to take upcoming changes into consideration. Furthermore, complex software systems often take a whole team of developers working in a long period of time to complete. This particular condition makes the coordination and the communication among team members essential in the success of the project. Therefore, one of the most crucial support for software development nowadays lies in documentation [11]. Indeed, a proper means of documentation not only provides necessary information about the system through development iterations but also facilitates the communication among stakeholders by providing a common understanding of the system. Of different aspects of a system to be documented, the architecture - a high level abstraction of the system structure - plays an important role. Architecture documentation is supposed to deliver the system's required functions as well as quality attributes [50]. It serves as a means to communicate the architecture to stakeholders so that they can learn, analyse, make decisions from it. Also, documenting software architecture during the early stage of development brings major benefits such as enabling early analysis, system visibility, complexity management, and enforcing design disciplines.

During the last decade, the research community on software architecture has tackled one important type of architecture documentation: architectural decision (AD) [22, 50, 27]. AD is considered to have big influence on the design of a software system. Either it is about the choice of a technology, a database or an applied pattern; AD significantly changes some parts of the system. A system change always starts off with an AD and eventually ends up by specific implementation. Having said that, AD documentation is often implicitly made because either it is a time-consuming task and thus, avoided by architects; or architects themselves do not see the immediate benefit from this extra effort. Nevertheless, explicit AD documentation is found to be necessary if not crucial in most software development processes. Indeed, the benefit of AD documentation is obvious since stakeholders do always need a deep understanding of the system's architecture: Developers want clear explications about the architecture to proceed with implementation. Customers want to make sure that the architecture satisfies their business requirements. Architects want to know the intention and the rationale of decisions that other architects have made. The lack of such explicit AD documentation can lead to design conflicts and eventually, the lost of the system's quality properties. This phenomenon is also referred to as the vaporization of architectural knowledge (AK) in the literature. To tackle this problem, there have been a lot of works aiming to provide a proper means of AD documentation that conveys rationale and supports the traceability of AD from the architectural model.

Of the commonly made ADs, those about the application of patterns are among the most popular ones [51, 4]. Patterns have been largely used in software design. They document existing, proven design experience in order to support the construction of software with well-defined properties. In the domain of software architecture development particularly, a typical design method is to select and combine a number of patterns that address the expected quality requirements and use them to build elements of the architecture [31]. Each element continues to be decomposed to more fine-grained elements using an appropriate pattern whenever possible. The process keeps going on until the elements at the lowest level are decided. This top-down design approach makes ADs about pattern use become central activities in architecture development. Construction and maintenance of architecture is then considered as the selection and enforcement of ADs about pattern. Another interesting characteristic of patterns, which make them the subjects of AD documentation, is that they are not simply design instructions to follow but also a blueprint of the architecture [9]. Basically, the structure of a pattern consists of elements that play certain roles and constraints imposed on them. Once this blueprint is applied on an architecture, every related architectural element must respect it. This structural aspect of patterns gives the potential to systematically document ADs about their application and automatically check possible conflicts with existing ADs.

Since patterns exist at different levels and granularities, a popular usage of patterns in the development of architecture is to combine patterns to create more complex ones [9, 2]. Pattern composition has been considered as a key requirement in software development and provide several benefits. First, in real world architectures recurring problems are complex and their solutions can be represented by patterns that require the combination and reuse of other ex-

isting patterns. The combined patterns on one hand, handle the increased complexity of the architecture and on the other hand, capture the properties of participating patterns. Second, in such environment, a given architectural element can be interwoven with different roles and constraints. Thus, combining patterns beforehand can provide early analysis and understanding about emerging properties of the design. Last but not least, considering pattern as a mere reusing structure, pattern composition gives architects the flexibility to customize and maximize its utility. Thus, in the literature, an important research topic is about designing a pattern language that supports easy, systematic pattern composition in a pattern-centric software development.

1.2 Problem statement

Current supports for structural AD documentation and verification are still at the early stage. While ADs that are well captured side by side with the architectural model do enable their recognition and traceability; when it comes to structural ADs, a more structural way of AD documentation that provides AD violation detection is still missing. The lack of this structural aspect of AD significantly limits the extent to which structural AD can be exploited.

Indeed, currently ADs are textually documented following certain templates, essentially providing textual information and explanations about how ADs are made and how to respect them. Little focus is paid on documenting AD's structural aspect which makes it difficult, if impossible at all, to verify AD's consistency. Without an automatic verification support, architects have to go through lengthy explanations to be able to proceed with the design, which is a tedious task. Moreover, this informal way of documenting AD could lead to ambiguity in understanding the design, which also makes it an error-prone task. This is where a structural approach of documenting AD comes to the rescue. In fact, not every AD is thoroughly structurally constructed and a such approach is not necessary. However, in an architectural development process where structural ADs, such as ADs about the application of patterns, are heavily leveraged, structural AD documentation approaches can be put in use.

On the way of selecting and modelling pattern constructs to capture ADs, we realized the need for pattern composition. Actually, existing pattern description approaches and pattern languages do provide constructs and mechanisms to compose patterns. However, pattern composition information is not adequately documented when composing operations are not part of the patter construct. Once the architectural solution achieved there is no means to know that it is a result of a composition of patterns. In other words, while the obtaining of a composed pattern is taken into consideration, the composition history is not. On one hand, this practice prevents the traceability of pattern composition, which significantly helps in pattern comprehension. On the other hand, it does not promote pattern customizability and reconstructability.

1.3 Aim of the thesis

The aim of this thesis is to use pattern constructs to provide an automated verification of AD violation, in particular structural ADs, and pattern-related ADs to be precise. Having also done a research on existing approaches about pattern description, we come up with a novel support for pattern composition. Thus, a pattern description language leveraging the concept of pattern composition is our another objective. These are two mainstream thoughts that are conveyed throughout the dissertation.

To complement the documentation of structural AD, we present a means for automatic checking of structural AD through the use of pattern constructs. We combine pattern formalization and AD documentation into an approach which has been shown to be complete in AD violation checking. Benefits of this approach are a convenient way of documenting structural AD and an efficient mechanism to detect conflicts in making AD.

To address the composition of patterns, we present a way of combining patterns that preserves composition information. Merging operators are treated as first-class status which has been shown to be important in pattern traceability and reconstructability. The benefit of this approach is an easy and customizable way of building composition-centered pattern catalogues.

With these two contributions, we believe that architects are provided with a better support for documenting and enforcing structural ADs. The support is shown to be especially significant in a pattern-centered architecture development process, where patterns are heavily used to leverage their knowledge.

1.4 Dissertation plan

Chapter 2 provides a brief background of concerning terminologies as well as an overview of the State of the art in two domains: AD documentation and pattern composition. Chapter 3 introduces COMLAN, a composition-centered pattern description language. We describe abstract and concrete syntax of the language and show how they are applied in a real-world architectural model. Chapter 4 presents the approach of using pattern as a means to document StADs and verify their consistency. This chapter comprises a general picture of the approach as well as detailed explanations on a concrete example. Chapter 5 introduces the architecture as well as the functionalities provided by two developed tools: COMLAN and ADManager, which realize our conceptual ideas. In each tool, we present use cases and how they are implemented via the tool. Chapter 6 shows two empirical evaluations for the two presented approaches, respectively. For each evaluation, we show the setup, the evaluating process, the analysis and the final result. Finally, in Chapter 7 we concludes the dissertation and open new perspectives.

2

State of the art

Contents

In this chapter we present an overview of the state of the art. We begin with a brief overview of software architecture, architectural decision and pattern, and then discuss work in two categories that are most closely related to our work: the documentation of architectural decision and the composition of patterns.

2.1 Background

Before proceed to the relevant literature about AD documentation, we would like to clarify the concept of AD and a special type of AD - AD about the application of pattern. We discuss these background knowledge in the sub-sections below.

2.1.1 Architectural Decision

The concept of architectural decision (AD) has been brought to the community of software architecture almost a decade ago since the date of this dissertation. Being one of the first efforts to describe AD, Jansen et al. [22] defined AD as the description of a set of any modification made to the software architecture together with its rationale, design rules and design constraints. More specifically, according to ISO/IEC/IEEE 42010 [20], AD is described as a concept that affects *architectural description elements*, pertains to one or more *concerns* and justifies *architectural rationale*. Correspondingly, an *architectural description element* could be a stakeholder, a viewpoint or a model element, etc. A *concern* could be any interest in the system such as behaviour, cost or structure, etc. Finally, an *architecture rationale* is the explanation, justification or reasoning about the architecture decisions that have been made and also about the architectural alternatives that are not chosen. This definition comes after many efforts in the literature to draw a complete structure for AD such as [22, 50, 56].

Because the architecture must endure a lot of changes upon which many different ADs are made, these ADs may contradict each other and thus result in design conflicts. That is the context where AD documentation comes to its utility as an important architecting activity. Indeed, AD documentation serves as a means to emphasize the rationale behind some design decisions having been made. Respecting these rationale makes sure that the architect can avoid possible design conflicts and the architecture evolves in harmony with existing design decisions. From this point of view, software architecture design is not only the matter of depicting a set of structures and their relation but also the result of making a set of ADs [22]. Moreover, since architecture is complexed to be communicated, capturing the rationale and why things are the way they are in the architecture facilitates the communication to future evolution [11].

2.1.2 AD about the application of pattern

During the software development, ADs can fall into many different categories and levels such as enterprise-affected ADs, project-affected ADs or technical ones [56]. Among them, structural ADs are supposed to be the most common ones [50, 39]. This is explained by the fact that every evolution of a system often begins with structural ADs. They modify the structure of architecture and affect the system on the highest level of abstraction.

In reality, a typical method of building the architectural structure of a system is to select and combine a number of patterns that address the expected quality requirements and use them to build elements of the architecture [3, 7]. Specifically, patterns are well-proven solutions

for particular recurring design problems that arise in a specific design context [9]. They serve many different purposes such as providing common vocabulary and understanding for design principles, a means of documenting software architecture, but most importantly supporting the construction of software with well-defined properties [16].

That being said, one of the most popular structural ADs are those concerning the application of patterns in the architecture. Indeed, patterns are the sources of some of the most important ADs and provide a rich set of architectural knowledge (AK) [54]. Moreover, reusable design knowledge normally documented in patterns can be adapted to inexpensively document AD in specific context [57]. Thus, the design of a pattern-centric architecture can be considered as applying successive pattern-related ADs that eventually result in its final structure [11].

2.2 AD documentation

In the literature there are many proposed models and tools supporting AD documentation. Among these works, we can mention some representative models such as the architectural decision template [50], the ontology of design decisions [27] or recently the MAD 2.0 model [52], and tools such as Archium [21], ADDSS [10], AREL [41]. Most of the proposed models focus on characterizing AD. They point out which decisions architect have to deal with and what important elements an architectural decision is made of. For instance, in [50] Tyree et al. argue that architects should make the decisions that identify the system's key structural elements. Figure 2.1 depicts the most important elements to document this type of AD. Some of them are: issue - the reason why the architect makes an AD, constraint - the arising constraint that the made AD poses to the system, related artifact - the related elements resulting from the AD, etc.

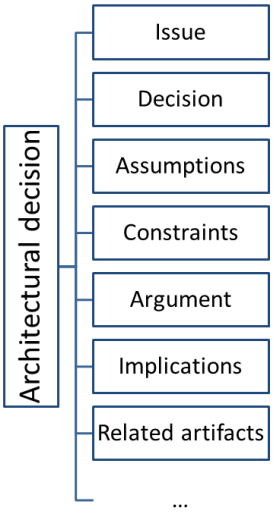

Figure 2.1: An AD template

On the other hand, proposed AD documentation tools try to figure out how to properly document and leverage AD. For instance, Archium [21] proposes to make AD along with the ADL model. Given the architectural model one should be able to trace back to the architectural decision it is based on and vice versa. This two-way traceability helps to maintain a better synchronization between the AD and its related architectural model elements. Figure 2.2 shows the meta-model of a software architecture from which Archium is implemented with first-class design decision [22]. The *Architectural Model* part comprises architectural elements while the *Design decision model* part comprises AD elements. Software architecture is described as a set of changes represented by the *Composition Model* part, which in turn include architectural model elements and AD elements. Therefore, a change does not only involve architectural model elements but also incorporate associated ADs. This idea of reserving first-class status for AD and documenting AD in parallel with architectural model elements has paved the way for many following works on AD modeling.

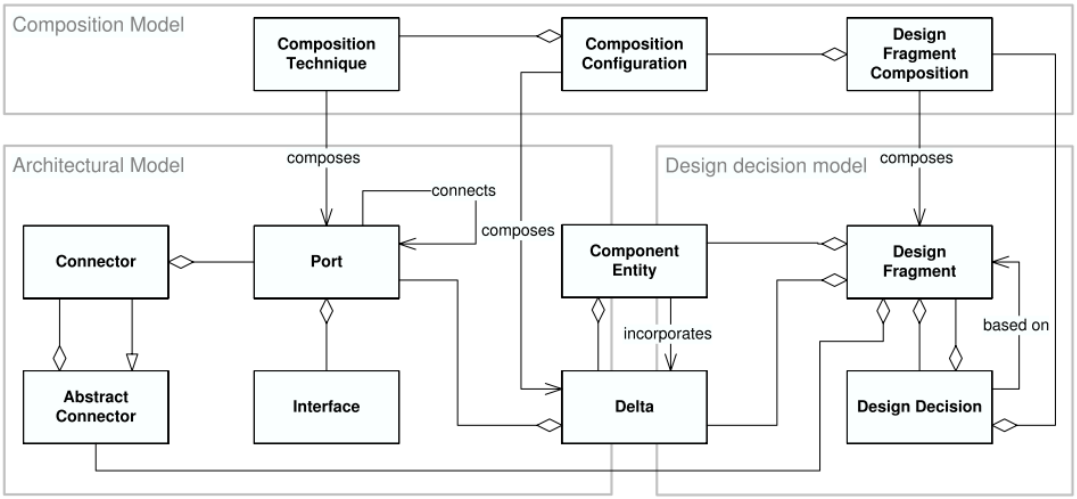

Figure 2.2: Meta-model of software architecture with first-class design decision (taken from [22])

Although these works have provided an efficient way to describe and document AD, architects still need to verify if an AD is respected or not by themselves. Given that the process of checking AD violation is error-prone, the automation of AD checking is a further step towards AD documentation.

2.2.1 Documentation of pattern-related AD

In [56], Zimmermann et al. point out the importance of reusable ADs in decision identification, decision making and decision enforcement and propose a model to document reusable ADs. In [57], they propose to weave pattern information into reusable architectural decision

models to benefit their mutual interests. Figure 2.3 depicts development stages of reusable AD models. At each stage of development, there exists a kind of pattern that corresponds to the reusable AD model. Analysis and architectural patterns correspond to ADs on the executive and conceptual level while design patterns fit in with the technology level, implementation and test patterns are used at the vendor asset level.

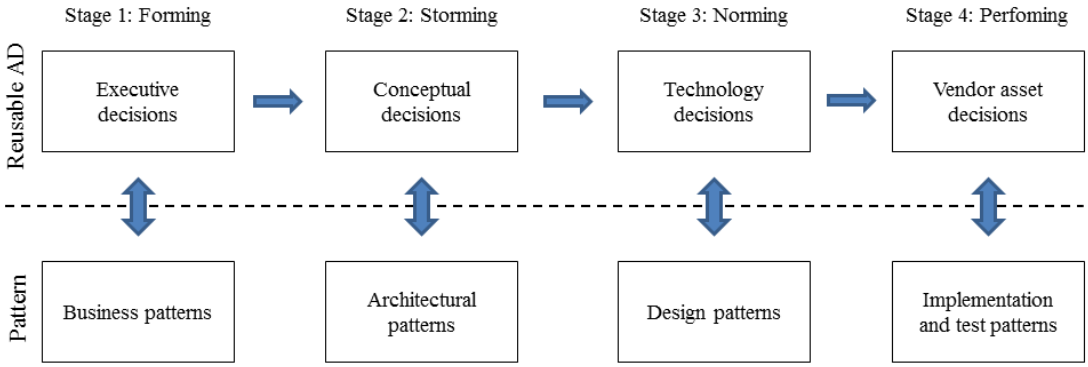

Figure 2.3: Development stages of reusable AD models (inspired from [57])

Besides that, in [19], Harrison et al. compare pattern and AD and think that the former can be leveraged to document the latter. These ideas focus on the fact that pattern use is an important information that completes the AD. Table 2.1 compares AD's features and those of patterns. Although patterns provide general, application-independent knowledge while ADs cover application-specific knowledge; they focus on the same issues such as context, solution alternative, rationale, etc.

2.2.2 Checking of pattern-related AD

Documentation of AD is obviously an inevitable architectural activity but, is it efficient enough to just literally write down ADs and expect architects to recognize them? It turns out that in complex architectures where many structural ADs are made, architects can mistakenly ignore or be unaware of ADs. This kind of mistake, what we call AD violation, can be avoided by a support of AD checking mechanism which will be explained in the followings.

2.2.2.1 Pattern Conformance

Patterns have become important constructs in existing ADLs such as Wright [1], Acme [17] or UML [15]. Patterns allow one to define a domain-specific design vocabulary, together with constraints on how that vocabulary can be used in constructing an architecture. In these ADLs, patterns are initialized by declaring instances of architectural elements and typing them with pattern elements. An architecture is said to conform to a pattern if there is no conflict between

AD	Pattern
Name	
Category	Group
	Status
Context	Assumptions, constraints
Solution variants	Positions
Rationale	Argument
Consequences	Implications
Known uses	
Related patterns	Related decisions
	Notes

Table 2.1: AD versus pattern (taken from [19])

pattern-typed architectural elements with respect to pattern constraints. In Acme and Wright, constraints are written based on first-order predicate logic language and pattern consistency is verified by formal specification checkers. The constraint 2.1 (taken from [1]) imposes that the architecture is designed with the star topology pattern. The constraint consists of two parts: i) The first part defines a component playing the "star" role which is connected to every connectors in the architecture. ii) The second part stipulates that the architecture is a connected graph; in other words, every components must be connected to at least one connector.

$$
\begin{aligned}
&\exists center : Component \bullet \\
&\quad \forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments \\
&\wedge \forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port \\
&\quad \mid ((c, p), (cn, r)) \in Attachments
\end{aligned} \tag{2.1}
$$

In UML, constraints, also known as well-formed rules, are written using OCL (Object Constraint Language) [36] and pattern consistency is referred to the conformance of model against meta-model. For instance, in [32] Medvidovic el al. use the stereotype of UML to model component-based software architecture. Concepts in UML such as class, association, interface, etc. are respectively stereotyped with concepts from component-based architecture such as component, connector, attachment, port, etc. The following is a snippet of OCL code to describe the connectivity of a Wright architecture.

```
self.modelElements->select(oclIsKindOf(Class) and stereotype = WrightComponent)
     ->forAll(comp | self.modelElements
          ->select(oclIsKindOf(Class) and stereotype = WrightConnector)
```

```
        ->exists(con | isAttached(comp, con)))
```

Specifically, for every class playing the role of Wright component, there exists at least a class playing the role of Wright connector that is attached to it [1].

2.2.2.2 Solution of structural AD conformance

Architectural patterns can be used as structural solutions of ADs (called StAD in the remaining of this dissertation). StAD not only conveys the intention of architects but also shapes the structure of the architecture according to its corresponding AD. Thus, a StAD-conformed structure within the architecture not only represents the intention of an AD but also, in terms of AK, is the indicator of the existence of AK. In the structural point of view, while existing in parallel with informal information of AD such as the context, the problem, the rationale, etc., a StAD is considered as any addition, subtraction, modification made to the structure of software architecture. An architecture is said to conform to a StAD if all of StAD-related elements prevail in the architecture [21, 25]. This understanding of StAD conformance implies two requirements: i) given an architectural element, one should be able to trace back to the architectural decision which it is based on and ii) the main consequences of an executed StAD, or the changed elements in the model due to that StAD in other words, must be preserved in the architectural model. In case of StAD about the application of pattern, StAD-related elements are in fact those playing roles in the applied pattern. Thus, the StAD conformance implies that the outcome of the application of that pattern must be documented.

There are basically two trends of work in the literature to enforce StAD conformance. The first one focuses on applying architectural constraints at the architectural level to impose StAD while the second one establishes explicit links between architectural model and StAD.

Tibermacine et al. [47, 48] propose a family of architectural constraint languages to describe the structural part of AD. They put forward an important remark that many structural ADs are often documented with constraints. Those constraints not only characterize AD but also help detect AD violation. For example, the following snippet of code (taken from [48]) illustrates a constraint for the *Pipes and Filters* pattern.

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

[1] isAttached is a function to check if a component is attached to a connector

```
->forAll(con:Connector|con.role
        ->forAll(r:Role|ACS.subComponent
                ->exists(com:Component|com.port
                ->exists(p:Port|(r in ACS.configuration.binding)
                        and ((p.kind = 'Input')
                             and (r.kind = 'Sink'))
                        or ((p.kind = 'Output')
                             and (r.kind = 'Source'))))))
and
ACS.configuration.isConnected
and
ACS.configuration.binding.role.connector
->AsSet()->size() = ACS.subComponent->size()-1
and
ACS.subComponent->forAll(com:Component|
                        (com.port->size() = 2)
                         and (com.port->exists(p:Port|
                                              p.kind = 'Input'))
                         and (com.port->exists(p:Port|
                                              p.kind = 'Output')))
```

This constraint is written in a standard profile which is supposed to be transformed to constraints in different target profiles corresponding to different ADLs. Specifically, it stipulates that in a *Pipes and Filters*-conformed architecture, all components must conform to Filter style (with Input and Output port), all connectors must conform to Pipe style (with Source and Sink style) and they must be bound together.

In another work [28], the authors propose to impose OCL constraints at model level to insure StAD. Many different off-the-shelf decision types are introduced, each one is represented by an OCL rule. Figure 2.4 shows an example of applying constraint on the architecture to document a StAD. The class Replication inherits the class Decision and additionally include an OCL constraint. This OCL constraint brings an additional support for StAD checking against AD-related architectural elements. Specifically, the OCL constraint states that the referenced component should be replicated *numberReplicas* times. This practice is known as the *Replication* pattern [46].

Another way of documenting StADs through pattern use is to construct StADs in form of linking elements between the StAD model and pattern-related elements in the architectural model. In [25], the authors propose to reinforce the outcome of StAD by using model differences. The outcome of StAD is considered as a set of model changes. Model changes serve as bindings between affected model elements and model differences. The architectural model is consistent with made StADs if and only if affected model elements prevail. Figure 2.5 illustrates the meta-model showing the binding between StAD and design model. Elements on the right side represent the AD while those on the left side represent model elemetns. These former are linked to these latter via model difference elements. Specifically, a *ModelChange* represents any kind of operation made to the architecture (addition, deletion, modification) which

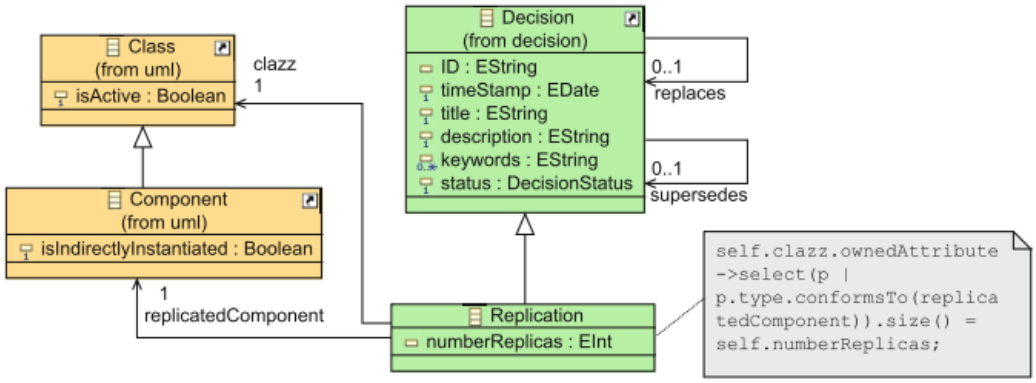

Figure 2.4: An AD imposing the replication pattern (taken from [28])

results in affected *ModelElement*. Via *ModelDifferences*, they are contained in the *Outcome* of an AD and then, used to verify the conformance of the changed architecture against a given AD.

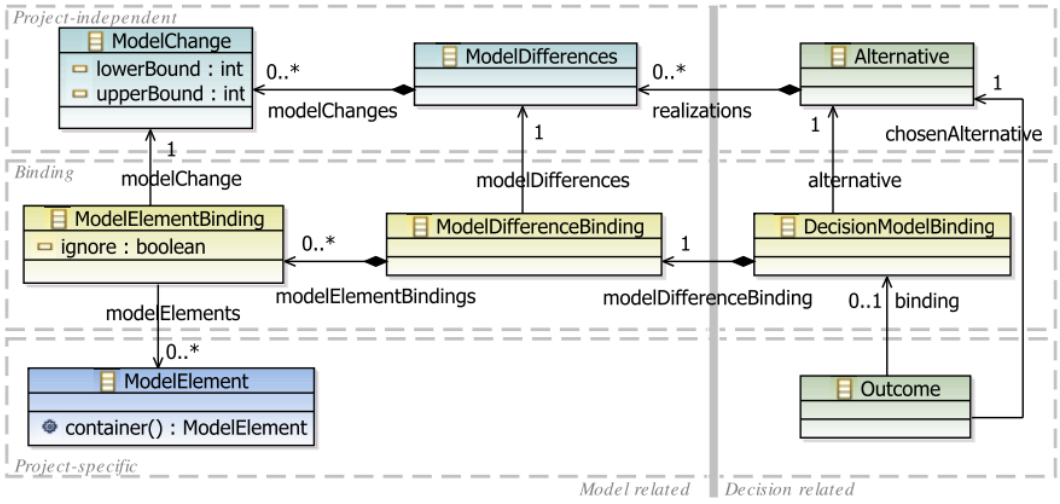

Figure 2.5: The binding meta-model of AD and design model (taken from [25])

Similarly, in [30], the authors influence the outcome of StAD by using actions. A StAD is provided with a set of actions which in turn are concretized into design model elements. Checking rules are automatically derived from the actions via a transformation mechanism. For example, the following snippet of code (taken from [30]) lists a set of action to impose the Indirection pattern [2].

```
compound indirection (cv A B) {
  add component "${A}${n}"
```

```
  add port "${A}${n}_I1" kind=PROVIDED to ${cv}.${A}${n}
  add port "${A}${n}_I2" kind=REQUIRED to ${cv}.${A}${n}
  add port "${A}_I" kind=REQUIRED to ${cv}.${A}
  add port "${B}_I" kind=PROVIDED to ${cv}.${B}
  add connector "${A}_I_${A}${n}_I1" from ${cv}.${A}.${A}_I to ${cv}.${A}${n}.${A}${n}_I1
  add connector "${A}${n}_I2_${B}_I" from ${cv}.${A}${n}.${A}${n}_I2 to ${cv}.${B}.${B}_I
  add stereotype <<"${n}">> to ${cv}.${A}${n}
}
```

Specifically, in the context of a component view (*cv* in the above rule), it adds an indirection component and two connectors from this component towards the target component (A) and the client (B). This set of actions, once used to generate architectural elements, also imposes the outcome of StAD. The common point of these two works is that reusable StAD is represented by a set of changes in the architectural model. The architectural model is said to be consistent with StAD as long as these changes prevail.

2.3 Architectural pattern modelling languages

Firstly, it is worth mentioning that in the literature, the term architectural style is used slightly differently from architectural pattern. The latter is the solution to a specific problem while the former does not require a problem for its appearance [11]. However, they both are structural idioms for architects to use. Since we only focus our interest on the structural aspect of these idioms, throughout this thesis architectural pattern is used as an interchangeable term for both of them.

In the literature there have been some efforts to model architectural patterns and their properties. For instance, there are work focusing on the use of formal approach to specify patterns. In the Wright ADL [1], the authors tend to provide a pattern-oriented architectural design environment where patterns are formally described. Similarly, Acme ADL [17] uses the term family for the specification of the family of systems or recurring patterns. The system is then instantiated from the definition of the family. For example, the following is an excerpt of Acme description of pattern and its instantiation.

```
Family PipeFilterFam = {
Component Type FilterT = {
Ports { stdin; stdout; };
Property throughput : int;
};
...
}

System simplePF : PipeFilterFam = {
Component smooth : FilterT = new FilterT
...
}
```

The Family PipeFilterFam is defined with a Component Type FilterT. It is then instantiated in the System simplePF with a component named *smooth* typed with FilterT. This declaration allows the system to make use of the types in the family, and it must satisfy all of the family's invariants.

As opposed to these domain specific languages, in [32] the authors propose to use general purpose languages such as UML to model architectural patterns. The approach consists of incorporating useful features of existing ADLs by leveraging UML extensions. More specifically, stereotypes on existing meta-classes of UML's meta model are used to represent architectural elements and OCL (Object Constraint Language) is used to ensure architectural constraints. The approach has been shown to be able to model different ADLs such as C2, Wright and Rapide.

In [53], the authors propose to use a number of architectural primitives to model architectural patterns. Through the stereotype extension mechanism of UML, one can define architectural primitives to design a specific structure of a pattern. In particular, those primitives are not only common structure abstractions among architectural patterns but also demonstrations of the variability in each pattern. Figure 2.6 is the example taken from [53] of the Broker pattern. The Broker consists of a client-side Requestor to construct and forward invocations, and a server-side Invoker that calls the target peer's operations. The Request Handler forwards request messages from the Requestor to the Invoker. The Request Handler component can only be accessed by the Requestor or the Invoker, no other components are allowed. This limit of accessibility is realized via an architectural primitive called Shield as shown in Figure 2.6. This primitive is also applied in other patterns such as Layer, Façade, etc. which makes it a common composable structure in pattern solutions.

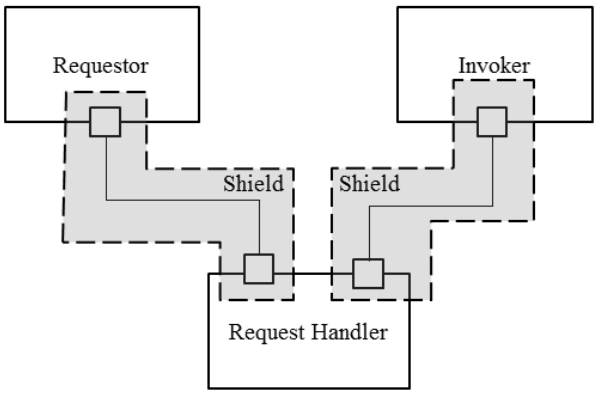

Figure 2.6: Shield architectural primitive

2.4 Pattern composition

Pattern composition is an important technique to deal with the complexity of pattern description and the combination of pattern's properties. There are mainly two branches of work on the composition of patterns. The first including [18, 5, 38] proposes to combine patterns at the pattern level which means that patterns are composed before being initialized in the architectural model. These approaches support two types of pattern element composition. The first type consists of creating a totally new element which is the product of the unification of participating elements. Regardless of different terminologies used in [18] (conservative composition), in [38] (unification) or in [5] (overlapping), the same idea is that the combined element will have all the characteristics of participating elements, and these will no more be present in the combined structure. An example taken from [38] is the composition of the Mediator pattern and the Proxy pattern as shown in Figure 2.7. The composition takes place between the Colleague class of the Mediator pattern and the Real subject class of the Proxy pattern. In its original pattern, the Colleague class extends the Colleague Interface. Similarly, the Real subject class extends the Subject class and contains the Request method. In the combined pattern, the Real Colleague class, which is the product of the composition of Colleague class and Real subject class, inherits all the features of its constituent classes. More specifically, it is a Real subject that can communicate with other Colleagues of a Mediator structure.

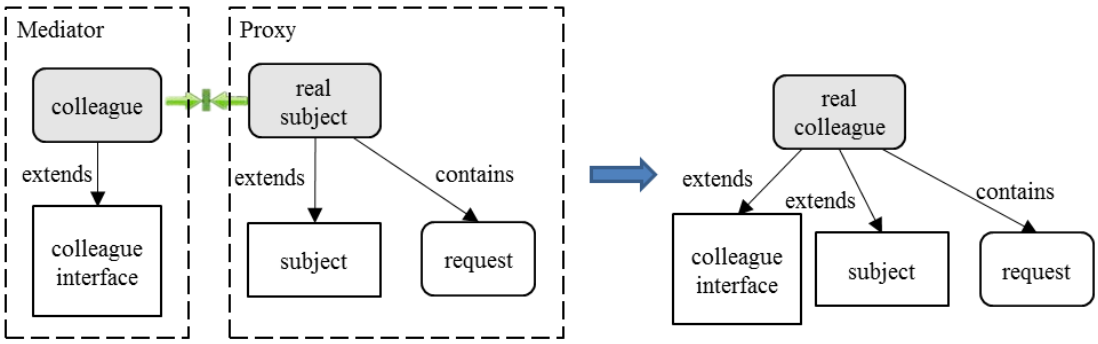

Figure 2.7: Overlapping composition of Mediator pattern and Proxy pattern

The second type implies that the participating elements in the composition keep their own identity, no new structure is formed because of the composition. Instead, a link element is added to connect participating elements. This composition is called combinative composition in [18] or conjunction in [38]. The example of the composition of the Mediator pattern and the Proxy pattern is retaken to illustrate this type of composition. As shown in Figure 2.8, the composition takes place between the Mediator class of the Mediator pattern and the Proxy class of the Proxy pattern. The result of this composition is an added Proxy Reference which connects the Mediator and the Proxy.

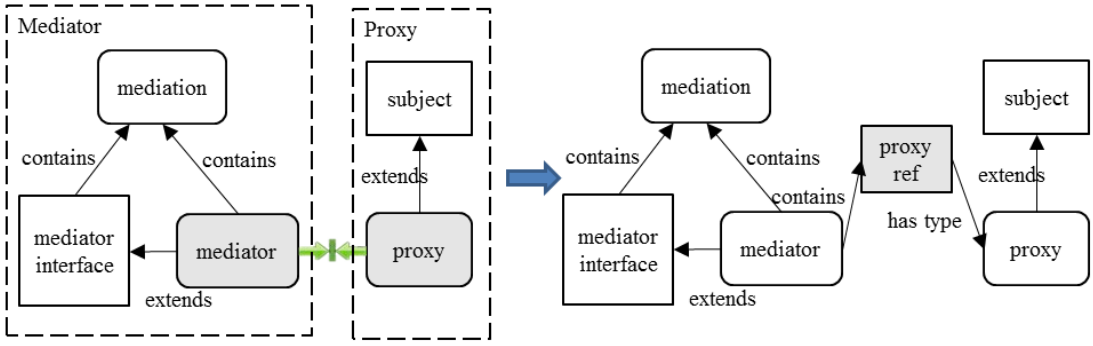

Figure 2.8: Conjunction composition of Mediator pattern and Proxy pattern

On the contrary to the approaches above, in [12], Deiters et al. propose to compose pattern at instance level. An architecture entity can at the same time play roles from different architectural building blocks which in fact represent architectural patterns. As a result, the affection of different architectural building blocks to an architecture entity is not only an instantiation but also a composition.

In another work [23], Jing et al. propose a UML profile to attach pattern-related information on merged elements in composed patterns. Figure 2.9 is an example taken from [23]. The Business Delegate pattern is composed with the Adapter pattern by overlapping the Business Delegate class and the Adaptee class. As we can observe, the overlapped element Business Delegate is annotated with the following tagged value:

```
<<{BusinessDelegate@BusinessDelegate[1]}{Adaptee@Adapter[1]}>>
```

This annotation indicates that the class plays two roles at the same time, one is Business Delegate from the Business Delegate pattern and the other is Adaptee from the Adapter pattern. Therefore, the constituent patterns can be traced back from the composed pattern. Similarly, [13] proposes different types of annotations, such as Venn diagram-style, UML collaboration, role-based tagged pattern, to make design pattern identifiable and traceable from its composition with others.

Patterns can also be expressed via architectural constraints. The composition of patterns is thus realized by the composition of architectural constraints. In [49], Tibermacine et al. propose to model architectural constraints by components. Constraints are represented by customizable, reusable and composable building blocks. As a result, higher-level or complex constraints can be built thanks to the composition of existing ones. Figure 2.10, which is taken from [49], is the example of the Pipes and Filter pattern constraint component. This component is internally composed by other components, each of them represents an architectural constraint such as the restriction of port and role, the connectivity of participating components, etc.

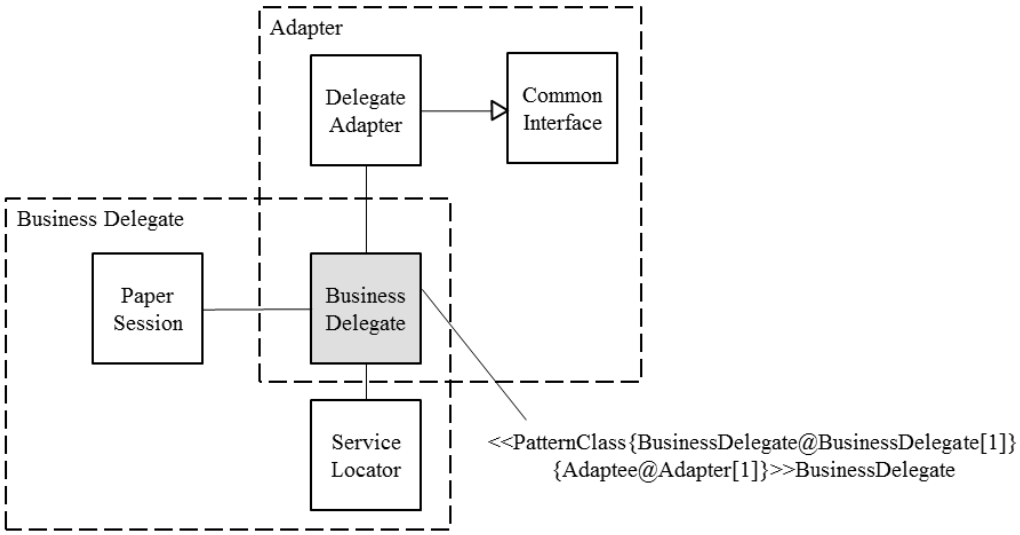

Figure 2.9: UML profile for attaching pattern composition information

2.5 Limitations of existing works

In the previous sections, we have shown that there have been a lot of attentions in the research community on the documentation of AD and structural AD in particular. We have also shown the importance of using pattern as a construct to formalize structural AD. Moreover, when it comes to pattern description, the composition of patterns is a key activity. Nevertheless, even though these approaches have illustrated their usefulness in the documentation of structural AD, they are still incomplete when detecting AD violation. In terms of pattern composition, current approaches also do not allow to fully exploit composition information. In this section we point out the shortcomings of the state of the art via some illustrative examples. These examples are chosen from case studies documented in the literature. In each example we show how the existing approaches cannot fulfil certain requirements. Then, we show how we develop our proposal to address these issues.

2.5.1 The problem of StAD maintenance and checking

To illustrate the need of maintaining StADs and how existing work cannot respond to this need, we take the FRC (Forestry Regulatory Commission) case study which is described in [46]. FRC is dedicated to commercial activities related to the forestry industry. It has been expanded in the past and expects to continue to change which results in costly development. Therefore, SOA (Service Oriented Architecture) has been opted to help the system respond more quickly to changing requirements. During the transition step to SOA solution, instead of rebuilding existed components, it was considered faster and less expensive to reuse them. It

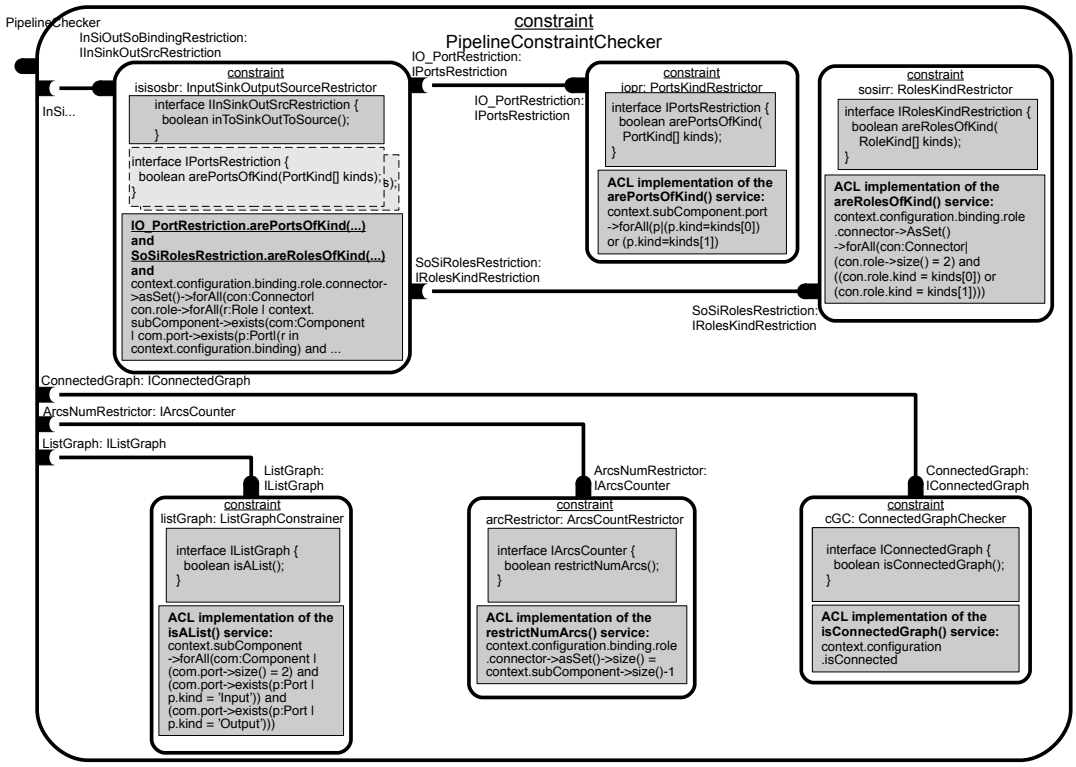

Figure 2.10: Architectural constraint composition (Figure taken from [49])

is the case of the *Fines service* and the *Evaluations service* which want to access to different repositories managed by a legacy component called *Data Controller*, a standalone Java EJB (Enterprise Java Beans). The *Legacy Wrapper* pattern [46] was chosen to handle this situation as illustrated in Figure 2.11.

More specifically, the *DWSA Data Service* which is deployed as a web-service is added to wrap the legacy component *Data Controller* to assure a seamless communication. The advantage of this pattern is that it allows the *Data Controller* component to perform changes and refactoring efforts without affecting the other service consumers that bind to it. This pattern implies the constraint stipulating that every service can only connect to the *Data Controller* component via the wrapper *DWSA Data Service*. The application of the Legacy Wrapper pattern in the FRC architecture is a StAD that needs to be taken into consideration throughout the evolution of FRC. In the viewpoint of the existence of related elements, this StAD is considered to be preserved as long as the legacy component *Data Controller* and the wrapper *DWSA Data Service* exist in the architecture of FRC. On the other hand, in the viewpoint of the pattern's structural consistency, this StAD is maintained as long as the legacy component *Data Controller* is the only component being able to access to the wrapper *DWSA Data Service*.

Later, a new service called *Appealed Assessments* was added to FRC and it also needs to

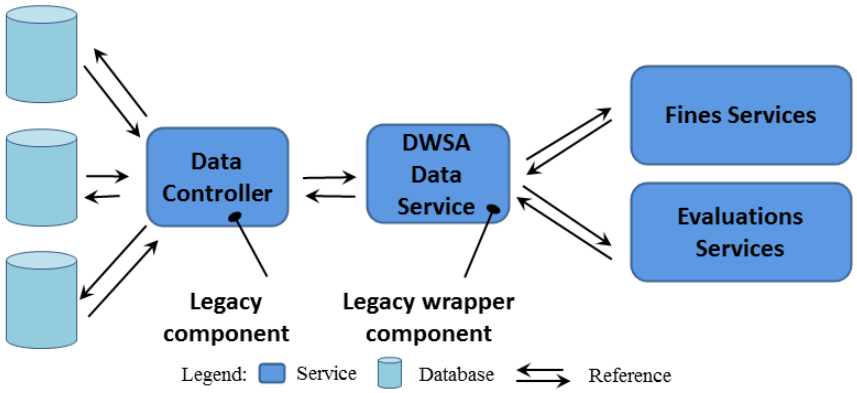

Figure 2.11: Legacy Wrapper pattern in FRC

access the *Data Controller* component. The architect that decided this addition was not completely aware of the rationale behind the existence of the wrapper *DWSA Data Service*. He then decided to use the Service Façade pattern [46]. More specifically, a façade called Data Relayer is added inside the *Appealed Assessments Service* with the only purpose to communicate with the component *Data Controller* (Figure 2.12). The reason influencing the architect not to use the Legacy Wrapper pattern is that Service Façade is simpler to implement, although the service using a façade will be coupled to the legacy component.

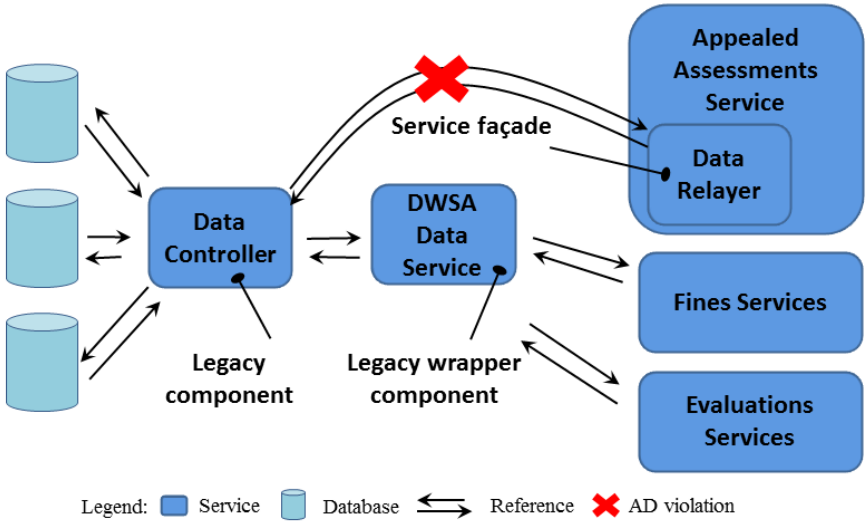

Figure 2.12: Architectural decision violation by adding Service Façade pattern

As we can observe in Figure 2.12, the constraint of the *Legacy Wrapper* pattern is violated due to the fact that the Appealed Assessments Service can still connect to the *Data Controller* component via the *Data Relayer* façade without passing through the wrapper *DWSA Data*

Service. In spite of the existence of the legacy component *Data Controller* and the wrapper *DWSA Data Service*, the decision of using Legacy Wrapper pattern has been violated since its structural consistency is not insured. This example shows that the existence of StAD-related elements is a necessary condition but not a sufficient one to detect the violation of StADs. Indeed, one indispensable part of architectural pattern is constraints imposed on future evolution of concerned elements.

Once informed about the violation, the architect changed the Legacy Wrapper pattern to a less rigid version which allows façade components to connect to legacy components. Thus, the link between *Appealed Assessments Service* and *Data Controller* is no longer a violation. Later, another architect participated in the project. He was not aware of the extended version of the Legacy Wrapper and he found that the *Appealed Assessments Service* must not access the *Data Controller* component directly. He deleted the link from the *Appealed Assessments Service* component towards the legacy component *Data Controller* and the *Data Relayer* façade as well. Then he added a link between the *Appealed Assessments Service* component and the legacy wrapper *DWSA Data Service*.

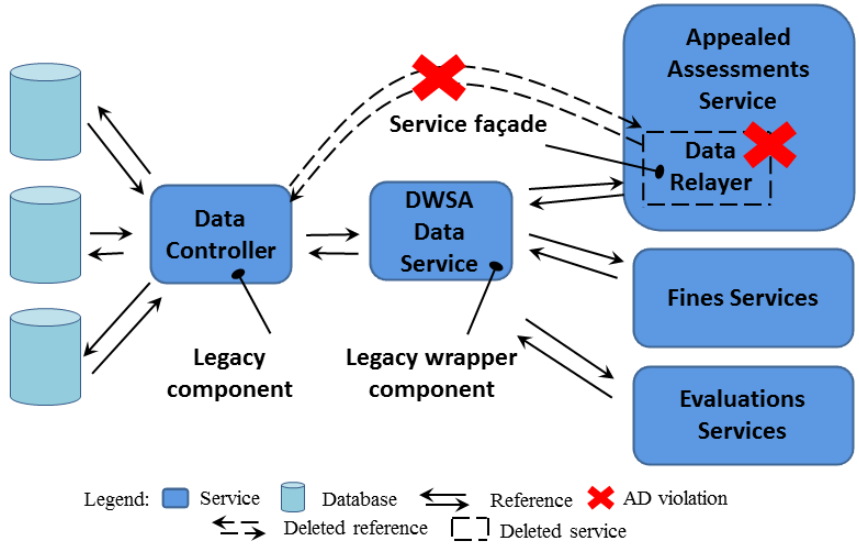

Figure 2.13: Architectural decision violation by deletion

Despite that the structural consistency of Legacy Wrapper pattern is always insured (there is no direct access to the legacy component), the decision of allowing the *Data Relayer* component (façade) to access directly to the *Data Controller* component (legacy component) as an extended version of Legacy Wrapper pattern has not been preserved as shown in Figure 2.13. Thus, the structural consistency of pattern is also a necessary condition but not a sufficient one to detect the violation of StADs. Indeed, the obvious prerequisite of an architecture conformed to a given pattern is that the concerned elements must exist.

In summary, these examples show that the documentation of StADs about the application

of patterns should focus on two aspects: the existence of related elements and the structural consistency of the applied patterns. Moreover, they are complementary aspects and both of them must be considered in evaluating StADs. In other words, one aspect can not replace the role of the other one and vice versa. The lack of one of these two aspects could lead to undetected StAD violations. In the literature, the existing work about ADLs [1, 17] or architectural constraints [47, 48] focus solely on the structural consistency aspect. The validity of StADs is maintained as long as concerned elements structurally conform to their playing roles in the architecture. Whereas, the other works about ADs [25, 28, 30] on the other hand, focus on the existence aspect. A StAD is considered to be preserved as long as modifications to concerned elements persist in the architecture.

2.5.2 The problem of pattern composition

Architectural patterns tend to be combined together to provide greater support for the reusability during the software design process. Indeed, architectural patterns can be combined in several ways. We consider here three types of combination: A pattern can be blended with, connected to or included in another pattern. To highlight the existing problems, we first show an example for each case of architectural pattern composition and then point out issues drawn from them.

2.5.2.1 Blend of patterns

By observing the documented patterns in [9, 11], we can see that there are some common structures that patterns share. For example, the patterns *Pipes and Filters* and *Layers* share a structure saying that their elements should not form a cycle.

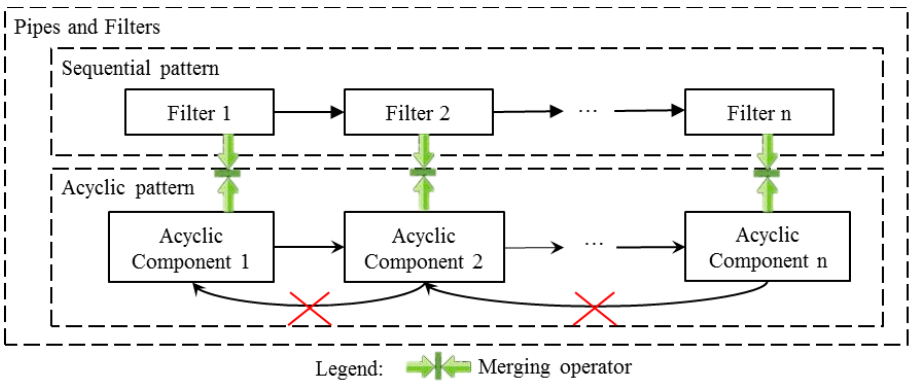

Figure 2.14: Pipes and Filters

If we consider to express the constraint that no circle can be formed from filters via a pattern, we can say that the pattern *Pipes and Filters* is composed of two sub-patterns (see Figure 2.14). We call them *Sequential pattern* and *Acyclic pattern*. The former consists of

Filter components linked together by *Pipe* connectors and the latter consists of *Acylic components* in a way that no cycle can be formed from them. Thus, *Pipes and Filters* is actually the product of the blend of these two patterns. But unfortunately, it is impossible to reuse the *Sequential pattern* or the *Acyclic pattern* alone because they are completely melted in the definition of the *Pipes and Filters* pattern. For instance, considering the construction of another variant of *Pipes and Filters* where cycles among Filters are accepted, it is beneficial to reuse the *Sequential pattern*.

2.5.2.2 Connection of patterns

A lot of documented patterns that are formed from two different patterns can be found in [11, 2]. One of these examples is the case where the pattern *Pipes and Filters* can be combined with the pattern *Repository* to form the pattern called *Data-centered Pipeline* as illustrated in Figure 2.15.

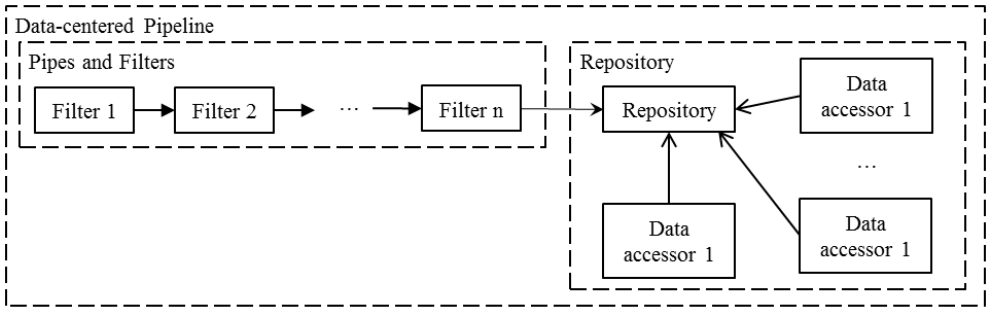

Figure 2.15: The Data-centered pipeline pattern

As we can observe, the two patterns are linked together by a special connector which serves two purposes at the same time: convey data from a *Filter* and access to the *Repository*. But once the composed pattern built, it is even more difficult to identify and reuse the sub-patterns in its constituent patterns. For instance, the fact that *Pipes and Filters* is the product of the composition of two sub-patterns is hardly noticeable.

2.5.2.3 Inclusion of patterns

Another type of architectural pattern composition is the situation when architectural patterns themselves can help to build the internal structure of one specific element of another pattern. In [2], we can find several known-uses of this type of pattern composition. An example where the *Layers pattern* becomes the internal structure of *Repository pattern* is shown in Figure 2.16. Indeed, when we have to deal with data in complex format, the *Layers* pattern is ideal to be set up as the internal structure of the repository since it allows the process of data through many steps. Moreover, the inclusion of patterns can be found at different levels. To be able to model such case, it is necessary to recursively explore patterns through many levels.

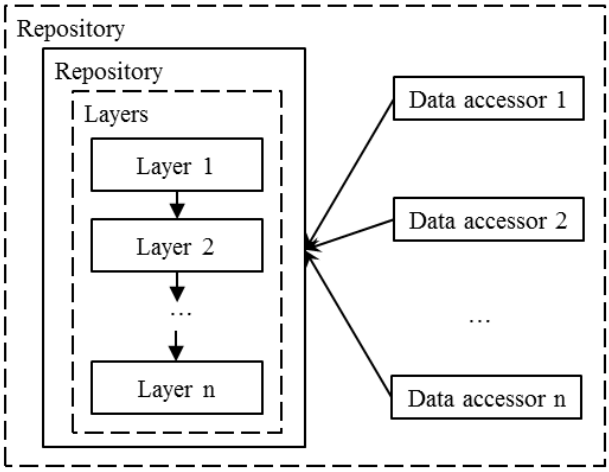

Figure 2.16: Layers as internal structure of Repository

2.5.2.4 Discussion

As we can observe from the example of subsection 2.5.2.2, the *Pipes and Filters* pattern is used as a constituent pattern to build the *Data-centered pipeline* pattern. When we look at the *Pipes and Filters* pattern in this view, we have no idea that it is composed from other patterns as shown in Example 2.5.2.1. We think the fact that the border between constituent patterns of a composed pattern is blurred can reduce greatly the pattern comprehensibility. Moreover, since the composed patterns may be then used to build another pattern, we believe that the traceability, which is the ability to know the role and the original pattern of every element in the pattern, becomes really essential.

Another issue to be taken into consideration is the reconstructability of composed patterns. In the example of subsection 2.5.2.1, when one of the two pattens forming the *Pipes and Filters* pattern changes, we should be able to propagate the change to the *Pipes and Filters* pattern. Moreover, since the *Pipes and Filters* pattern has been changed, the *Data-centered Pipeline* in which it participates in Example 2.5.2.2 must be also reconstructed. The same requirement exists for the example of subsection 2.5.2.3. For example, when another Layers pattern variant is used to form the internal structure of the Repository component, the change should be reflected in the composed pattern.

As shown in Section 2.4, in the literature, the already proposed approaches about pattern composition present pattern merging operators in an *ad-hoc* manner where information about the composition of patterns is vaporized right after the composition process. Thus, they ignore two aforementioned issues. Although in [23], one can trace back the constituent elements from which an element is composed, a composition view showing how the original patterns are composed is still missing and moreover, the support for reconstruction is ignored.

In summary, the examples shown above highlight two problems to solve:

1. *Traceability of constituent patterns*: One should be able to trace back to constituent patterns while composing the new pattern.

2. *Reconstructability of composed patterns*: Any time there is a change in a constituent pattern, one should be able to reuse the merging operators to reflect the change to the composed pattern.

2.6 Summary and discussion

This chapter sums up existing works in the state of the art in two main categories: AD documentation and pattern composition. In the former category, we highlighted the focus of the community on the first-class status of AD in the software development process. The documentation of AD has been proved to bring many benefits. Of these benefits, one most important is a clear vision about the rationale of the AD, which conveys certain quality properties of the architecture. We also pointed out the important role of patterns in the documentation of reusable AD. On one hand, patterns are central artefacts in attribute-driven architectural design process. On the other hand, patterns are shown to have similar characteristics to AD which can be leveraged to complement AD documentation. However, AD documentation is not the only concern of architects. Once supplied with well documented AD, they still need a support for detecting AD violation. Indeed, especially when it comes to structural AD, it may be easy to understand the created AD, but not always obvious to maintain a conformed, coherent architectural structure, as known as the solution of structural AD (StAD). To preserve StAD and avoid any kind of violation, related works in the literature propose two basic methods. The first one focuses on applying architectural constraints and architectural patterns at architectural level to impose StAD while the second one establishes explicit links between architectural model and StAD. Even though both kinds of approach show the ability to detect StAD violations, each one of them does not detect all violations. This remark has been highlighted via the example of the evolution of an architectural model in Section 2.5.1. We believe that the two approaches are complementary aspects to detect StAD violation and they should coexist to complete one another. Thus, one main part of this thesis is to promote the combination of architectural patterns and explicit architectural model-StAD links to enable StAD violation checking. Please also note that we do not attempt to promote another means of documenting AD, otherwise we focus on the automation of AD violation checking, which is an error-prone process without further support.

The second category of state of the art is about the composition of patterns. In fact, the need of composing patterns also derives from the subject of AD documentation. When it comes to pattern-related AD, chances are we could come across structures that involve many patterns at once. Thus, AD documentation involves the use of patterns that are composed from other patterns. We have showed that no matter of the type of pattern representation (constraint, model) or the type of composition (stringing, overlapping), existing works in the

literature do not consider composition operators as a part of pattern's elements. They exist in an *ad-hoc* manner in the sense that what really matters is the resulted composed pattern but not the composition process. Via different examples of pattern composition in Section 2.5.2, we pointed out that the this way of composing pattern does not allow the traceability and reconstructability of patterns. Thus, another part of the thesis is to propose a pattern description language that preserves the first-class status for merging operators.

The following chapters of the thesis aim at presenting these two above ideas, illustrate them through application examples and evaluate them with real data.

3

COMLAN - COMposition-centered pattern description LANguage

Contents

The need of a pattern description language comes from our intention to capture pattern-related StADs. While examining possible pattern-related StADs we found out that compound patterns are often applied in the architecture. They are patterns that can be built by combining more fine-grained patterns. Instead of building a compound pattern from scratch, one can select some unit patterns and combine them. In this chapter we present a pattern description language that favours composition by leveraging merging operators. We present the language in general and show its application with examples.

3.1 Process Overview

We propose the process of constructing patterns including two steps as illustrated in Figure 4.2. The first step consists in describing a pattern as a composition graph of unit patterns using the COMLAN language. In this presentation form, the pattern comprises many blocks, each block represents a unit pattern, all linked together by merging operators.

The second step consists in refining the composed pattern in the previous step by concretizing the merging operators. More specifically, depending on the type of merging operator, a new element is added to the composed pattern or two existing elements are mixed together. On the purpose of automating the process of pattern refinement, we use the Model Driven Architecture (MDA) approach [35]. Each pattern is considered as a model conforming to the COMLAN meta-model (see Section 3.2) in order to create a systematic process thanks to model transformation techniques. Thus, each refined pattern is attached to a corresponding pattern model from step 1 and any modification must be done only on the latter at step 1. At this stage, we offer the architect a pattern description language based on the use of classical architectural elements, architectural patterns and pattern merging operators.

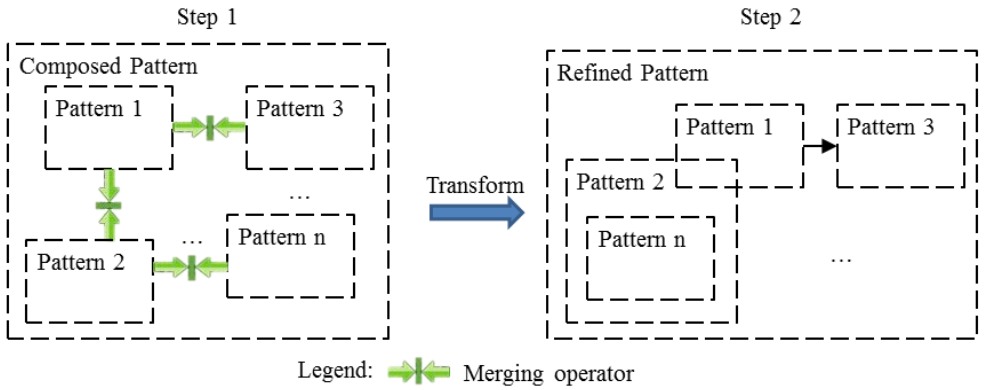

Figure 3.1: Overall Approach

We can see that through this two-step process, anytime we want to trace back the constituent patterns of a composed pattern in the second step, we can find them in its corresponding pattern model. Thus, we solve the traceability problem pointed out in the previous section.

We solve the second problem (reusability of merging operators) by the fact that merging operators are first-class entities in our pattern description language. In other words, merging operators are treated as elements of the pattern language where we can manipulate and store them in the pattern model like other elements. Therefore, the composition of patterns is not an *ad-hoc* operation but a part of pattern. This proposal facilitates significantly the propagation of changes in constituent patterns to the composed pattern. Indeed, the latter can thoroughly be rebuilt thanks to the stored merging operators. So, merging operators not only do their job which performs a merge of two patterns but also contain information about the composition

process. Thus, we think documenting them is one important task that architects should take into consideration.

In the following sections, we describe our pattern description language through its meta-model followed by the description of a pattern model and then, the transformation process that produces the refined pattern model from the pattern model.

3.2 The COMLAN meta-model

We introduce COMLAN as a means to realize two main purposes: build complex patterns from more fine-grained patterns using merging operators and leverage hierarchical patterns. The language only emphasizes the structural solution of patterns, thus patterns that are based on behavioural aspects of an architecture are not supported. As shown in Figure 3.2, our meta-model is composed of two parts: the structural part of the architecture and the pattern part. As pointed out in [24, 1] and also described in [11], the design vocabulary of an architectural pattern necessarily contains a set of component, connector, port and role. We take these concepts into consideration to build the structural part of our language. More specifically, they are described in our language as follows:

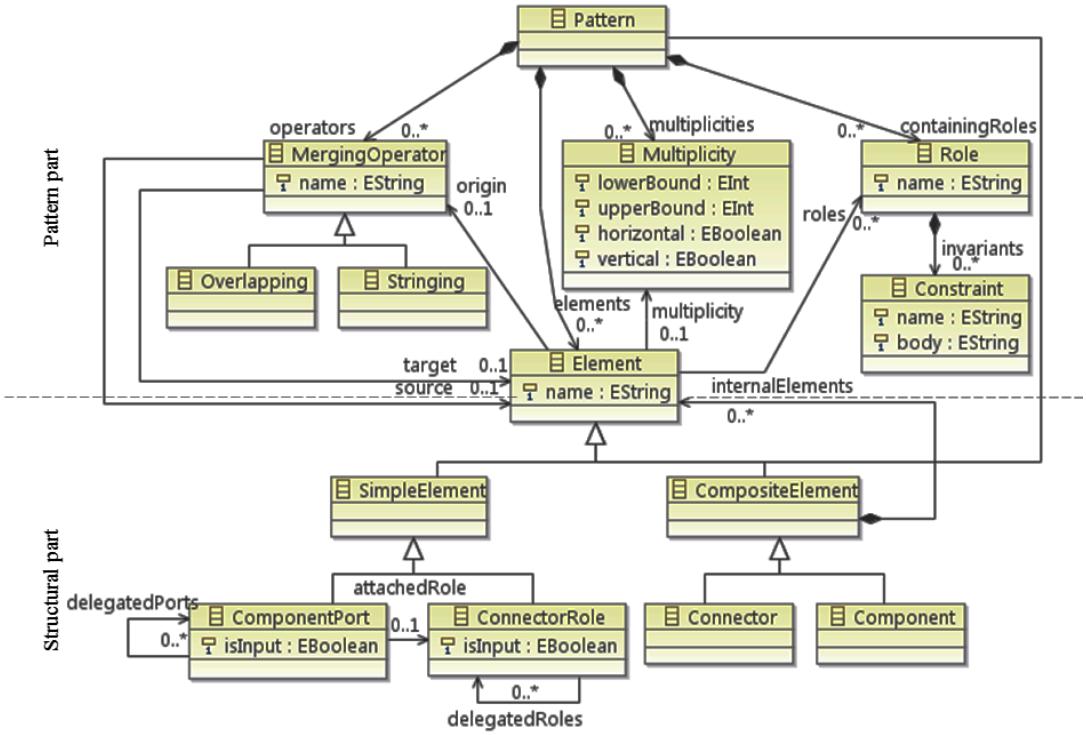

Figure 3.2: The COMLAN meta-model

- *Component* is a composite element which, through the *internalElements* relation, can contain a set of component ports or even a sub-architecture with components and connectors. These two types of containment relation are differentiated by a constraint imposed on the meta-model.
- *Component port* is a simple element through which components interact with connectors. A component port can be attached to a connector role or delegated to another component port in an internal sub-architecture.
- *Connector* is a composite element which, through the *internalElements* relation, can have a set of connector roles or even a sub-architecture with components and connectors. Similarly to the case of component, thanks to a constraint on the meta-model, these two types of containment relation are distinguished.
- *Connector role* is a simple element that indicates how components (via component ports) use a connector in interactions. A connector role can be delegated to another connector role in an internal sub-architecture.

The *pattern aspect* part (see Figure 3.2) of our meta-model aims at providing functionalities to characterize a meaningful architectural pattern. To be more specific, the meta-model allows us to describe a pattern element at two levels: generic and concrete. Via the *multiplicity*, we can specify an element as generic or concrete. A concrete element (not associated with any multiplicity) provides guidance on a specific pattern-related feature. Being generic, an element (associated with a multiplicity) represents a set of concrete elements playing the same role in the architecture. A multiplicity indicates *how many times* a pattern-related element should be repeated and *how* it is repeated. Figure 3.3 shows two types of orientation organization for a multiplicity: vertical and horizontal. Being organized vertically, participating elements are parallel which means that they are all connected to the same elements. On the other hand, being organized horizontally, participating elements are inter-connected as in the case of the pipeline architectural pattern [9].

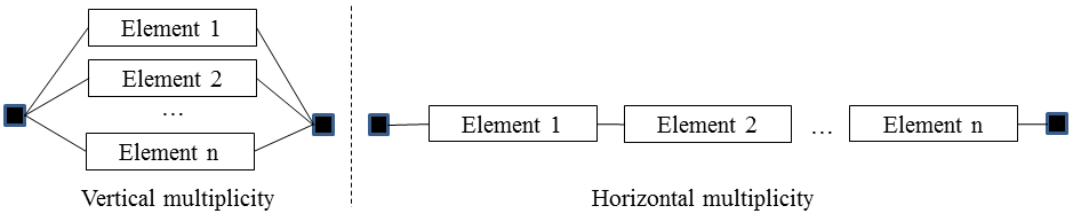

Figure 3.3: Orientation organization of generic elements

Each element in the meta-model can be associated with a *role*. A role specifies properties that a model element must have if it is to be part of a pattern solution model [15]. To characterize a role, we use architectural *constraints*. A constraint made to a role of an element helps to

make sure that the element participating in a pattern has the aimed characteristics. Constraints are represented in our approach in form of OCL [36] rules.

Similar to [18, 5, 38], in our language two types of merging operator are supported: stringing and overlapping as shown in Figure 3.4. As we can observe, these operators are preserved with firs-class status by being represented as model elements. A stringing operation means a connector is added to the pattern model to connect one component from one pattern to another component from the other pattern. If an overlapping operation involves two elements, it means that two involving elements should be merged to a completely new element. Otherwise, if an overlapping operation involves a composite element and a pattern, it means that the latter should be included inside the former. In both cases of merging, the participating elements are respectively determined through two references *source* and *target*. An element has an *origin* reference towards the merging operator from which it is concretized. This merging operator contains the information about the source element and the target element which allows the traceability of the composed element.

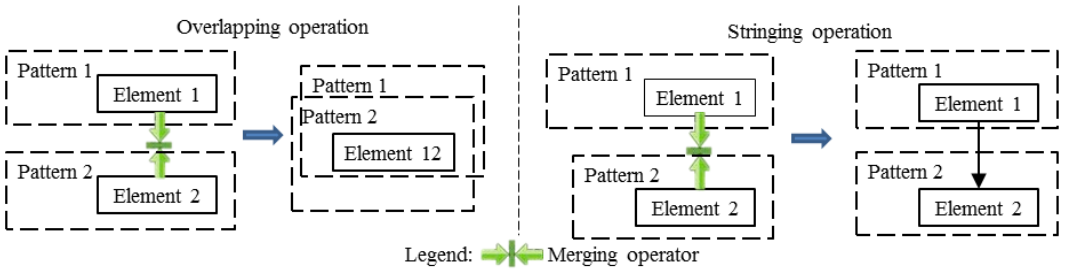

Figure 3.4: Two types of merging operation

Pattern can contain all concepts described above and most importantly, it inherits from *Element* which allows a composite element to contain it. This special feature helps our language to include an entire pattern into an element while constructing a pattern. In other words, hierarchical patterns are supported.

3.2.1 Example of pattern definition

For the purpose of illustration, our pattern definition language will be used to model an example about the pattern for data exploration and visualization as in the Vistrails application's architecture [8]. More specifically, this model represents the first step of the pattern definition process. As shown in Figure 3.5, this pattern model consists of four main sub-patterns: *Pipes and Filters*, *Client-Server*, *Repository* and *Layers*, all connected together through merging operators. Among these three patterns, the *Repository* pattern is a hierarchical one whose the component of the same name includes the *Layers* pattern.

To explain how the pattern concepts are realized, we go into details for the *Pipes and Filters* pattern. On the upper left corner of Figure 3.5, we can observe that the *Pipes and Filters*

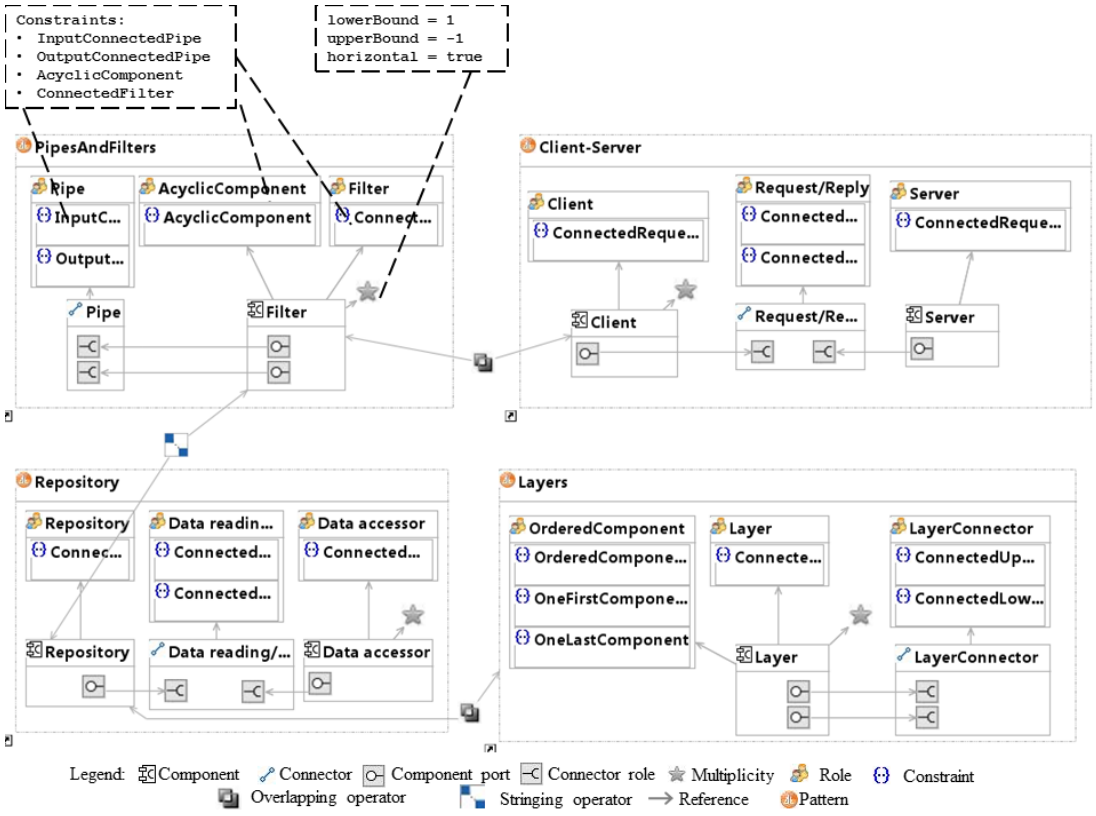

Figure 3.5: Example of pattern model

pattern is constructed with the emphasis on the following elements: the component *Filter* specified with two roles *Filter* and *AcyclicComponent*, the connector *Pipe* specified with the role *Pipe*. The connector *Pipe* is not assigned with any multiplicity. Otherwise, the component *Filter* is assigned with a multiplicity since it represents many possible filters inter-connected by Pipe connectors. Furthermore, its horizontal multiplicity[1] indicates that there may be many instances of Filters and they must be horizontally connected. The role *Filter* is characterized by the *ConnectedFilter* constraint. To be more specific, it stipulates that a filter cannot stand alone, there must be at least one pipe connected to a filter. Similarly, the constraint *AcyclicComponent* characterizing the role *AcyclicComponent* stipulates that among filters, we cannot form a cycle. Finally, the two constraints *InputConnectedPipe* and *OutputConnectedPipe* say that for a given pipe, there must be a filter as input and a filter as output. The above constraints are presented as OCL invariants as follows:

```
invariant AcyclicComponent:
if role->includes('AcyclicComponent') then
   Component.allInstances()->forAll(role = 'AcyclicComponent' implies not
```

[1] upperbound = -1 indicates that there's no limited upper threshold for a multiplicity

```
   self.canFormCycle())
endif;

invariant ConnectedFilter:
if role->includes('Filter') then
   Connector.allInstances()->exists(role = 'Pipe' and isConnected(self))
endif;

invariant InputConnectedPipe:
if role->includes('Pipe') then
   Component.allInstances()->exists(role = 'Filter' and
   getOutputConnectors().contains(self))
endif;

invariant OutputConnectedPipe:
if role->includes('Pipe') then
   Component.allInstances()->exists(role = 'Filter'
   and getInputConnectors().contains(self))
endif;
```

Merging operators are used to link participating patterns together. More specifically, in our pattern model (see Figure 3.5), three merging operators are used:

- An overlapping operator whose source is the *Filter* component in the *Pipes and Filters* pattern and target is the *Client* component in the *Client-Server* pattern.
- A stringing operator whose source is the *Filter* component in the *Pipes and Filters* pattern and target is the *Repository* component in the *Repository* pattern.
- An overlapping operator whose source is the *Repository* component in the *Repository* pattern and target is the Layers pattern.

These three operators are used as elements of the pattern language and stored along with the other elements.

This example has shown the ability of using our language to describe complex patterns which are combined from different patterns by leveraging merging operators.

3.3 Pattern refinement

After being described as the composition of constituent patterns through merging operators, the pattern model will be refined. We consider the refinement as a model transformation where the source model is a pattern model with explicitly presented merging operators and the target model is a pattern model where merging operators are already concretized. Therefore, the transformation rules consist in processing merging operators in the composed pattern

model and produce appropriate results in the refined pattern model. While realizing this transformation, three important issues need to be taken into account: how to concretize a stringing operator, how to concretize an overlapping operator and how to handle nested patterns.

3.3.1 Stringing operator transformation

Among structural elements in the pattern language, except for components which can be linked by stringing operators, there is no interest to link together other elements like connectors, component ports or connector roles. That is the reason why a stringing operator can only be transformed into a new *connector* to link source component and target component. New component ports are also added to the source component and the target component and attached to new connector roles in the newly created connector. As shown in Figure 3.6, the stringing operator described in the previous step is now transformed to the connector *DataReading/WritingPipe*. This new connector contains two connector roles, one attached to a component port in the *ClientFilter* component and the other attached to a component port in the *Repository* component. A simplified version of the transformation algorithm for stringing operator is presented in Algorithm 1

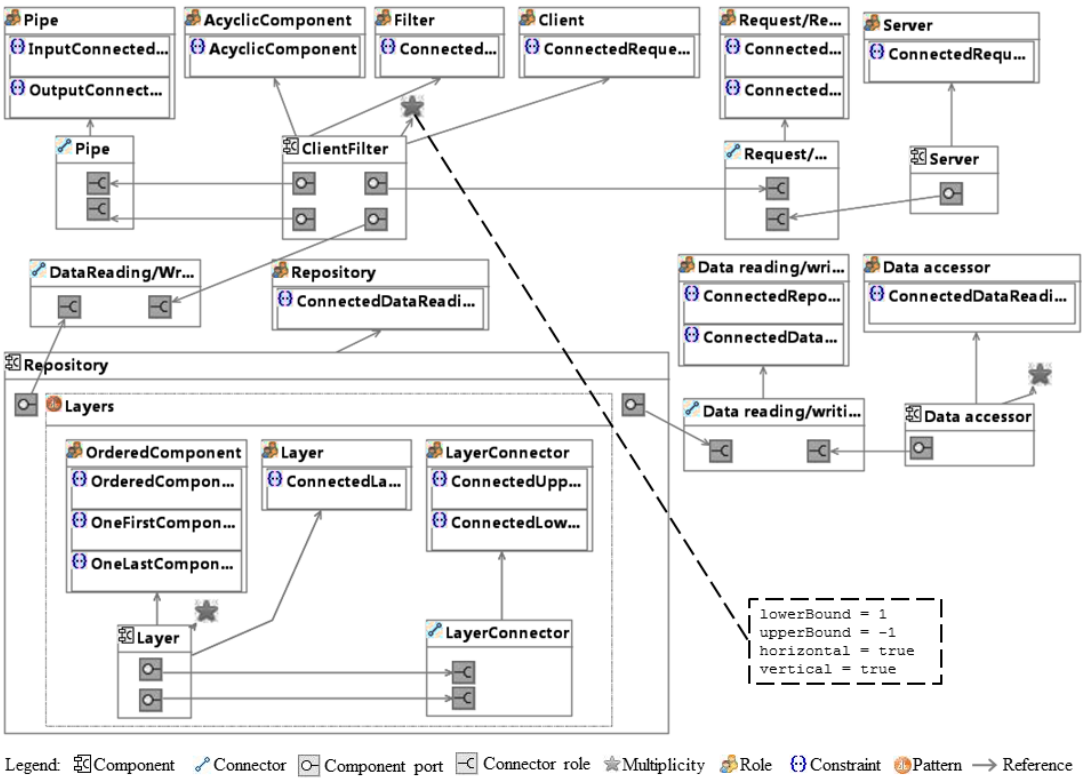

Figure 3.6: The refined pattern model

Algorithm 1 The stringing operator transformation algorithm

Require: Stringing *StrOp*
Ensure: Connector *Con*
1: Component $TarComp \leftarrow StrOp.target$
2: Component $SrcComp \leftarrow StrOp.source$
3: ComponentPort $SrcPort \leftarrow new\ ComponentPort$
4: ComponentPort $TarPort \leftarrow new\ ComponentPort$
5: ConnectorRole $SrcRole \leftarrow new\ ConnectorRole$
6: ConnectorRole $TarRole \leftarrow new\ ConnectorRole$
7: $TarComp.internalElements.add(TarPort)$
8: $SrcComp.internalElements.add(SrcPort)$
9: $Con.internalElements.add(SrcRole)$
10: $Con.internalElements.add(TarRole)$
11: $SrcPort.attachedRole.add(SrcRole)$
12: $TarPort.attachedRole.add(TarRole)$

The algorithm takes a stringing operator as input and produces a connector as output. First, the source and the target components are determined. Next, ports and roles to be attached to the source and the target component are respectively created. The final step consists of i) adding ports and roles to the appropriate components and the new connector, and ii) attaching roles to their corresponding ports.

3.3.2 Overlapping operator transformation

The result of the transformation for an overlapping operator is a new element which carries all the characteristics of the source element and the target element. For composite elements, the composition begins with the fusion of all internal elements. As we can see from Figure 3.6, the overlapping operator described in the previous step is concretized by the component *Client-Filter*. This component contains all component ports from the source element which is a *Filter* and the target element which is a *Client*. Furthermore, via these component ports, the link from the component to two connectors *Pipe* and *Request/Reply* is also preserved.

The overlapped element plays all the roles of the source element and the target element. Indeed, the *ClientFilter* plays three roles at once: *AcyclicComponent*, *Filter* since it participates as a *Filter* in the *Pipes and Filters* pattern and finally, *Client* since it participates as a *Client* in the *Client-Server* pattern.

The multiplicity is merged as follows: The range of the merged element's multiplicity is the intersection of that of the source element and the target element. More specifically, the merged lower value is the bigger one between the two lower values and the merged upper value is the smaller one between the two upper values. If the source element's multiplicity or the target element's multiplicity is vertical or horizontal then merged element's multiplicity is also vertical or horizontal. In our pattern model (Figure 3.6), the multiplicity of the merged component *ClientFilter* is both vertical and horizontal since its source component *Client* is vertical and its target component *Filter* is horizontal as illustrated in Figure 3.7. A simplified version of the transformation algorithm for overlapping operator is presented in Algorithm 2. The algorithm

takes an overlapping operator as input and produces a merged element as output. First, source and target elements are determined via the *source* and *target* references of the overlapping operator. Next, the source and target elements are tested whether they are composite elements. If it is the case, internal nested elements from the source and target elements are included in the newly created element. Finally, role and multiplicity elements are also respectively merged into the newly created element.

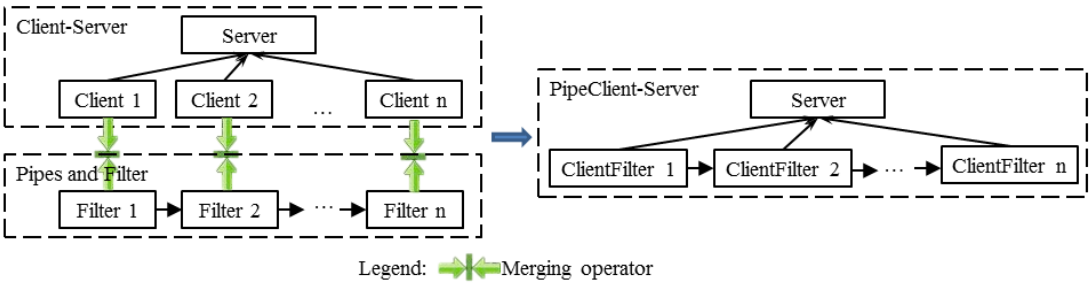

Figure 3.7: The merged pattern of Client-Server and Pipes and Filters

Algorithm 2 The overlapping operator transformation algorithm

Require: Overlapping $OvlOp$
Ensure: Element $MergedElem$
1: Element $TarElem \leftarrow OvlOp.target$
2: Element $SrcElem \leftarrow OvlOp.source$
3: **if** $TarElem$ isTypeOf $CompositeElement$ **then**
4: **for all** Element $e \in TarElem.internalElements$ **do**
5: $MergedElem.internalElements.add(e)$
6: **end for**
7: **end if**
8: **if** $SrcEle$ isTypeOf $CompositeElement$ **then**
9: **for all** Element $e \in SrcElem.internalElements$ **do**
10: $MergedElem.internalElements.add(e)$
11: **end for**
12: **end if**
13: **for all** Role $r \in TarElem.roles$ **do**
14: $MergedElem.roles.add(r)$
15: **end for**
16: **for all** Role $r \in SrcElem.roles$ **do**
17: $MergedElem.roles.add(r)$
18: **end for**
19: $MergedElem.multiplicity \leftarrow multimerge(SrcElem.multiplicity, TarElem.multiplicity)$

In the case of a chain of consecutive overlapping operators in which one continues another, we use Algorithm 3. Let us say we have n random elements linked together by (n-1) overlapping operators. The algorithm consists of n-1 steps. In the first step, the overlapping operator merges *Element-1* and *Element-2* to create *Element-12*. Next, *Element-2* is replaced by *Element-12*. In the second step, the overlapping operator merges the new *Element-12*

and *Element-3* to create *Element-123*. Similarly, *Element-3* is then replaced by *Element-123*. The algorithm continues so on until the $(n-1)$-th step when all elements are merged into the *Element-123..n*. An important remark in this algorithm is that thanks to the replacement mechanism, an element can reflect the merging operation in which it participates. Thus, the merging operation is propagated to every element participating in the merging chain. Notice that in the case of an overlapping operator between an element and a pattern, the former is always the source element and the latter is always the target element. This constraint is imposed in the meta-model. Thus, in a chain of overlapping composition, the pattern, if exists, always stays at the end of the chain.

Algorithm 3 The multi-overlapping transformation algorithm

Require: Set of Element $ElemSet$
Ensure: Element $MergedElem$
1: $n \leftarrow ElemSet.length$
2: **for** $i = 1$, i++, while $i < n$ **do**
3: $\quad MergedElem \leftarrow overlappingMerge(ElemSet[i], ElemSet[i+1])$
4: $\quad ElemSet[i+1] \leftarrow MergedElem$
5: **end for**

The algorithm takes a set of elements that are supposed to be merged together and produces the merged element. Next, a loop to each pair of elements is performed. In each iteration, a binary merge takes effect between two elements and assign the merged element to the element that participates in the next iteration.

3.3.3 Nested pattern transformation

If a pattern participates in a merging operation, all of its internal elements will be added in the refined pattern while the pattern itself will not be transformed. As shown in Figure 3.6, all the three patterns *Pipes and Filters*, *Client-Server* and *Repository* disappear leaving their internal elements in the refined pattern. Otherwise, if a pattern does not participate in any merging operation, a refinement procedure (which is actually a recursive procedure) will be applied to the pattern. Since the *Layers* pattern does not contain any merging operators, the refinement procedure just simply keeps all its internal elements.

3.3.4 Support of traceability and reconstructability

For every merged element in the composed pattern, the origin reference towards the merging operator (e.g. see the the *origin* reference from *Element* towards *MergingOperator* in the COMLAN meta-model)helps to preserve information about which element in the constituent pattern participating in the composition process. Figure 3.8 illustrates the support of traceability in the above example. The merged component *ClientFilter* containing a reference towards an explicit overlapping operator from which the source element Client and the target element Filter can be retrieved (e.g. via the *source* and *target* references respectively). Similarly,

thanks to references towards a stringing operator and an overlapping operator, the connector *DataReading/WritingPipe* and the component Repository respectively can trace back to their original source and target elements. The support for pattern reconstruction is pretty straightforward. Whenever a constituent pattern is modified, the composed pattern is updated with the changes automatically. Next, thanks to the stored merging operators, the refined pattern can be rebuilt taking into account the modifications. For instance, let us suppose the case that a more complex variant of the Layers pattern is used instead of its actual pure variant. Thanks to the overlapping operator between the *Repository* component and the *Layers* pattern, the new *Layers* variant can be reflected in the newly merged *Repository* component.

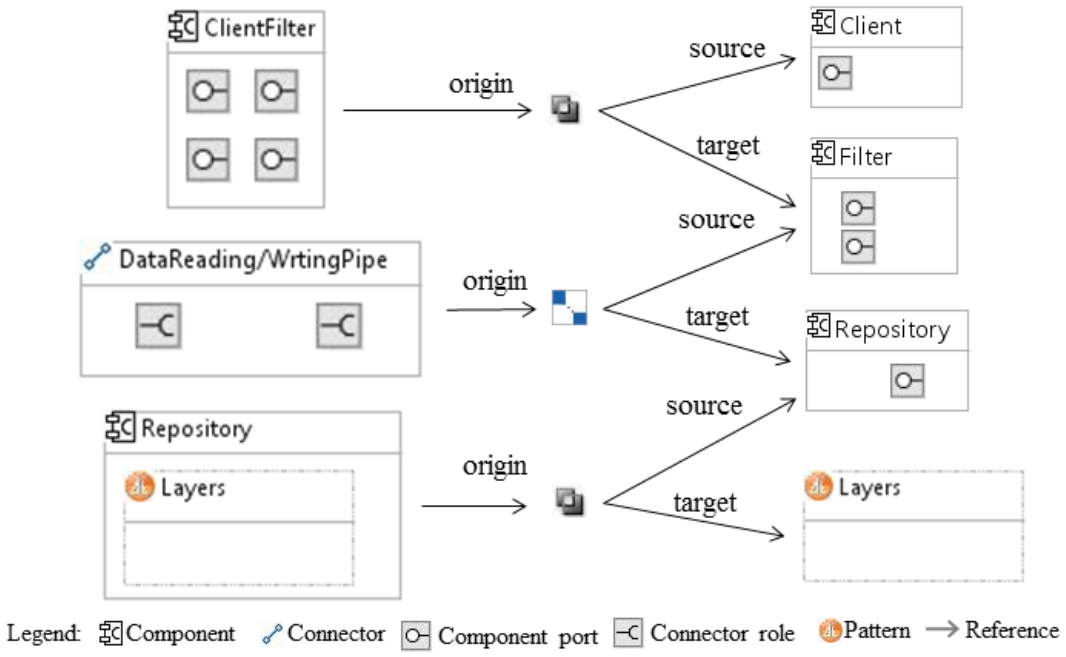

Figure 3.8: Support of traceability

3.4 Summary

In this chapter we presented a solution to the documentation of pattern composition information by leveraging the first-class status of merging operators. First, we gave an overall picture of the approach in general. Basically, the composition process passes through two steps. They are actually two different composition views of pattern. The first step provides a composition view in which each participating pattern keeps its own identity and patterns are linked from one to another via merging elements. The second step shows a concrete view in which merging elements are concretized into architectural elements. A special characteristic of this two-step process is that one can change from one step to another and vice versa. Changing from

the concrete view to the composition view can enable the traceability of the composed pattern. One can discover that a given pattern is composed from which pattern and the composition is realized using what type of merging operator. On the opposite sense, one can reconstruct a new variant of a composed pattern just by modifying participating pattern in the first step and propagate the changes to the second step.

We also showed the pattern description language in detail via its meta-model. Like other pattern description languages, the meta-model comprises necessary elements to model a full-fledged pattern. Except for these features, there are two main points that are worth mentioning about the meta-model. First, merging operators are modelled using meta-classes which means that they can be stored, manipulated, passed as reference, etc. like other elements in the meta-model. Though it makes the meta-model more verbose, the first-class status merging elements are shown to be efficient in the support of traceability and reconstructability of pattern. Second, the separation between the pattern part and the structural part brings the flexibility to language. One can keep the pattern part and switch the structural part to adapt from one architectural representation to another. To illustrate the usage of the language, we selected a representative example of an architecture in which four patterns coexist and are combined using two types of merging operators.

Finally, for each type of merging operator, we presented the refinement process in which merging operators are concretized into architectural elements in the composed pattern. While stringing operator eventually ends up in a new connector, overlapping operator triggers the grouping of participating elements.

The idea of using first-class merging operator in pattern composition has been presented in the ECSA conference [43] and the Future Generation Computer Systems journal [44].

4

Pattern-based approach for documenting the solution of structural architectural decision

Contents

In addition to common characteristics of a general AD, pattern-related ADs also convey the structural aspect of a pattern. This special feature makes it possible to verify the conformity of an architectural model against its applied pattern-related ADs thanks to StAD. StAD conformance basically implies two requirements: i) given an architectural element, one should be able to trace back to the architectural decision which it is based on and ii) the main consequences of an executed StAD, or the changed elements in the model due to that StAD in other words, must be preserved in the architectural model. As shown in Chapter 2, existing approaches that tempt to document StAD are shown to be incomplete to satisfy these two requirements. Therefore, in this chapter we present our own approach of StAD documentation. We present the approach in general and then show its application with examples.

4.1 General Approach

The main idea behind our work is that StADs about the use of patterns should be preserved throughout the evolution of the architecture. More specifically, the existence of pattern-related elements and the structural consistency of StADs about pattern use should be automatically checked whenever there is a modification to the architectural model. Because we only concentrate on StADs about pattern use and for the sake of simplicity, *the term StAD throughout the dissertation is understood as StAD about pattern use*. Moreover, we focus on the structural part of AD to support the conformance checking. It does not mean that the other parts of AD such as the rationale or the concerns are not important. Instead, together they make a complete structure of StADs that supports both the documentation and the automatic checking.

Similar to [56], StAD documentation in our approach goes through three steps: Pattern creation, StAD integration and StAD verification. Figure 4.1 depicts the process of using StADs in architecture construction.

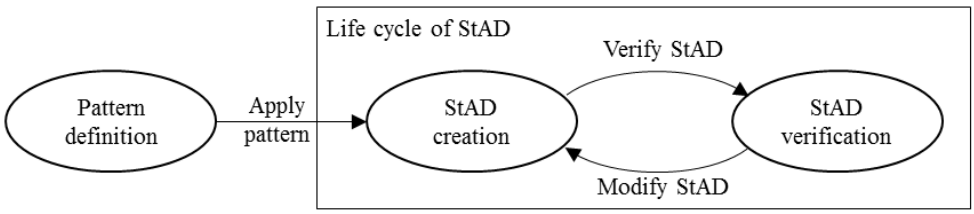

Figure 4.1: The process of using StAD

Pattern definition consists in the specification of a pattern structure. A pattern is defined once and used for all concerned StADs. *StAD creation* is the step in which the decision about the application of the defined pattern is created. Finally during the *StAD verification* step, the architectural model is checked whether it complies with the created StAD. Basically, the life cycle of a StAD begins to exist when a pattern definition is applied. Then, if the architectural model is found to be inconsistent with the created StAD during the StAD verification step, the architect can come back to the *StAD creation* step and modify or recreate another StAD and so forth.

On the purpose of automating the process of StAD documentation, we use the Model Driven Architecture (MDA) approach [35]. Each artifact is considered as a model conforming to its meta-model in order to create a systematic process thanks to model transformations and leverage existing MDA techniques (e.g. conformity verification).

In the remainder of this section, we will go further into each step in the StAD documentation process.

4.1.1 Pattern definition

We propose the use of a *general pattern language* for the purpose of pattern definition. As shown in Figure 4.2 (Pattern definition part), the abstract syntax of this language is described using a *general pattern meta-model* which contains only architectural elements involved in the pattern definition. These elements are determined through a survey of well-known architectural patterns such as those described in [46, 11, 9]. In terms of concrete syntax of our language, one can graphically define a meaningful architectural pattern in form of a *pattern model* using necessary elements. Furthermore, *pattern models* are also language-independent. With the separation between pattern definition and architectural design, no modification to the architectural model is needed to define a pattern, which makes it easy to adapt to different ADLs.

4.1.2 StAD creation

Links between pattern elements and their correspondent architectural elements play an important role in keeping track of StAD made to an architectural model. An explicit linking will facilitate the specification of StADs as well as their storage. In our approach, links between pattern elements and architectural elements are represented by *mapping models* (illustrated in the Pattern integration part of Figure 4.2). A *mapping model* indicates that a StAD has been applied on an architectural model.

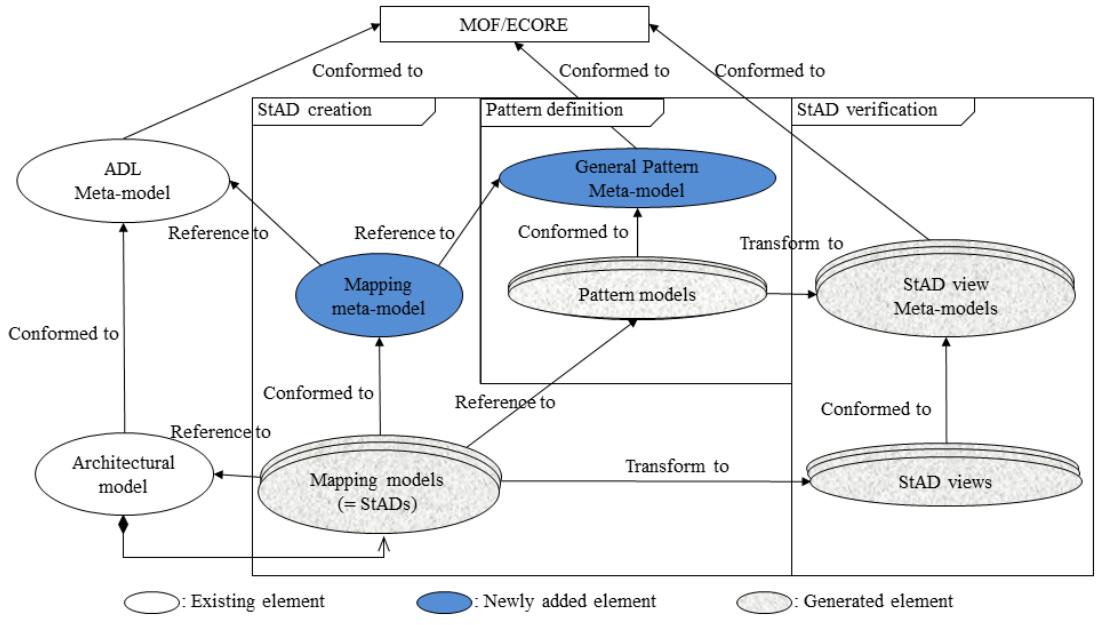

Figure 4.2: MDA approach for StAD documentation

In the literature, architecture is considered as a set of views which are representations of system elements and relations associated with them [11]. Each view serves a specific purpose

depending on the concerns of one or more stakeholders. Having taken this viewpoint into account, we propose to consider an *architectural model* as a multi-view representation where each view contains only elements related to a specific StAD. In this scenario, *StAD views* are filtered from the *architectural model* through a transformation mechanism in which *mapping models* are the source models and *StAD views* are the target models. In fact, we can consider the *mapping models* as the initial version of the *StAD views* where information about the integration of StAD are stored. Thus, the transformation is the step in which these information are concretized into *StAD views* elements.

4.1.3 StAD verification

To make sure that an *architectural model* is consistent with created StADs, not only do the existence of StAD-related elements in an *architectural model* need to be verified but also the constraints imposed on them need to be handled. To achieve the first goal, the presence of StADs in the *architectural model* is checked through the completeness of *mapping models*. Indeed, *mapping models* are intermediary bridges between the *architectural model* and the *pattern model* and thus, the incompleteness of *mapping models* shows the lack of StADs in the *architectural model*. To achieve the second goal, the constraints imposed by patterns on the *architectural model* are checked through the consistence of *StAD views*. To check the conformity of *StAD views*, we chose to first transform the *pattern models* into *StAD view meta-models* (Pattern verification part in figure 4.2) and then, make use of model checking techniques from MDA [35].

4.2 Pattern definition

The process of creating a pattern consists in instantiating a *pattern model* from its meta-model. We first introduce the *general pattern meta-model* from which pattern models are created. Then, we clarify the pattern definition process through a concrete example.

4.2.1 *General pattern meta-model*

The COMLAN meta-model consists of two parts: the structural part and the pattern part. While the pattern part is general enough to be applied in any paradigm, the structural part represents concrete elements for each supported language family. Throughout this chapter, we intentionally opt for Service Oriented Architecture (SOA) as the language family. Therefore, the structural part contains all necessary architectural features from SOA and concepts related to SOA pattern definition. As we can guess, the pattern part is kept unchanged. Figure 4.3 illustrates this SOA-adopted pattern language.

Inspired by the SCA model[1] [6], we construct the structural part of our *General Pattern*

[1] SCA is a model created by a group of industrial partners to support building applications and systems using SOA solution.

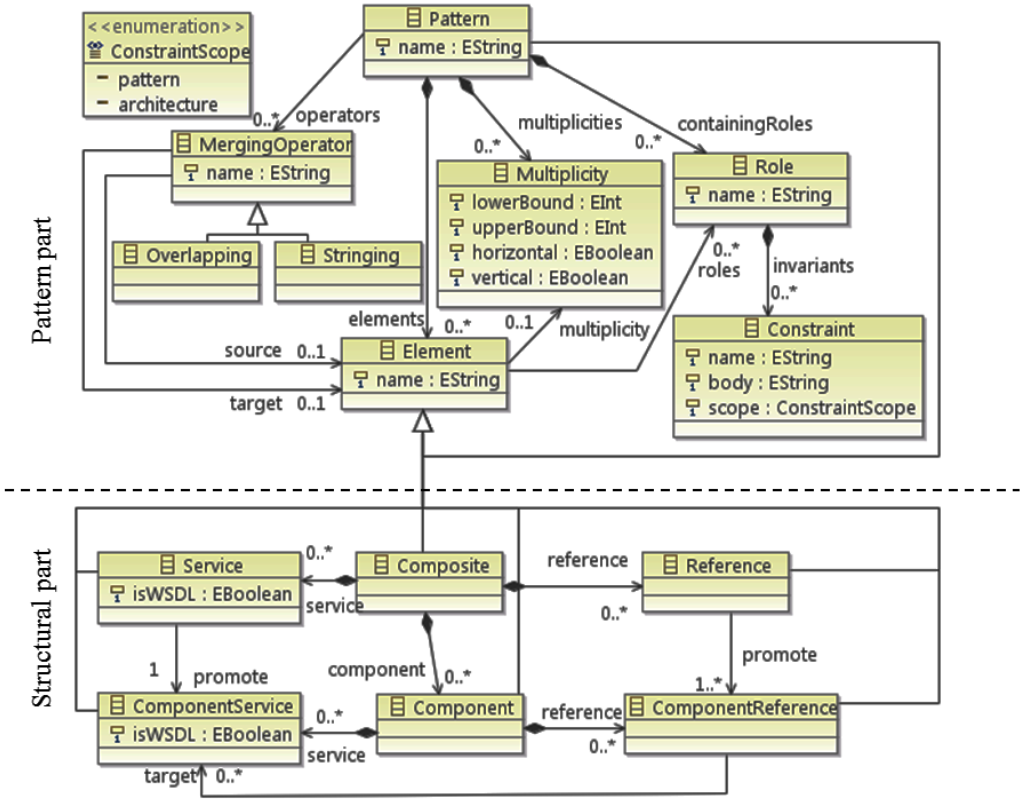

Figure 4.3: SOA General Pattern Meta-model

Meta-Model for the SOA description language family as follows:

- *Composite* serves as the container to assemble and connect service-oriented building blocks together.

- *Components* are basic units of the architecture that represent business functions from which composite applications are built.

- A component is composed of *component services* and *component references*. The former provide functionalities supported by the component and the latter play the role of consuming services of other components. A component reference can be wired to a component service through its *target* attribute. The attribute *isWSDL* specifies whether the component service is a webservice or not.

- Thinking of composites as black-box components, they have also *services* and *references*. To be consumed by the world outside of a composite, a component service of the containing component can be promoted as a service of the composite. Similarly, to be

served by an outside service, a component reference should be promoted as a composite's reference.

The *pattern aspect* part of this meta-model aims at providing functionalities to characterize a meaningful architectural pattern. All the details can be found in Section 3.2.

4.2.2 Architectural Pattern Specification

For the purpose of illustration, we will examine the SOA Legacy Wrapper pattern [46] which is also mentioned in Section 2.5.1.

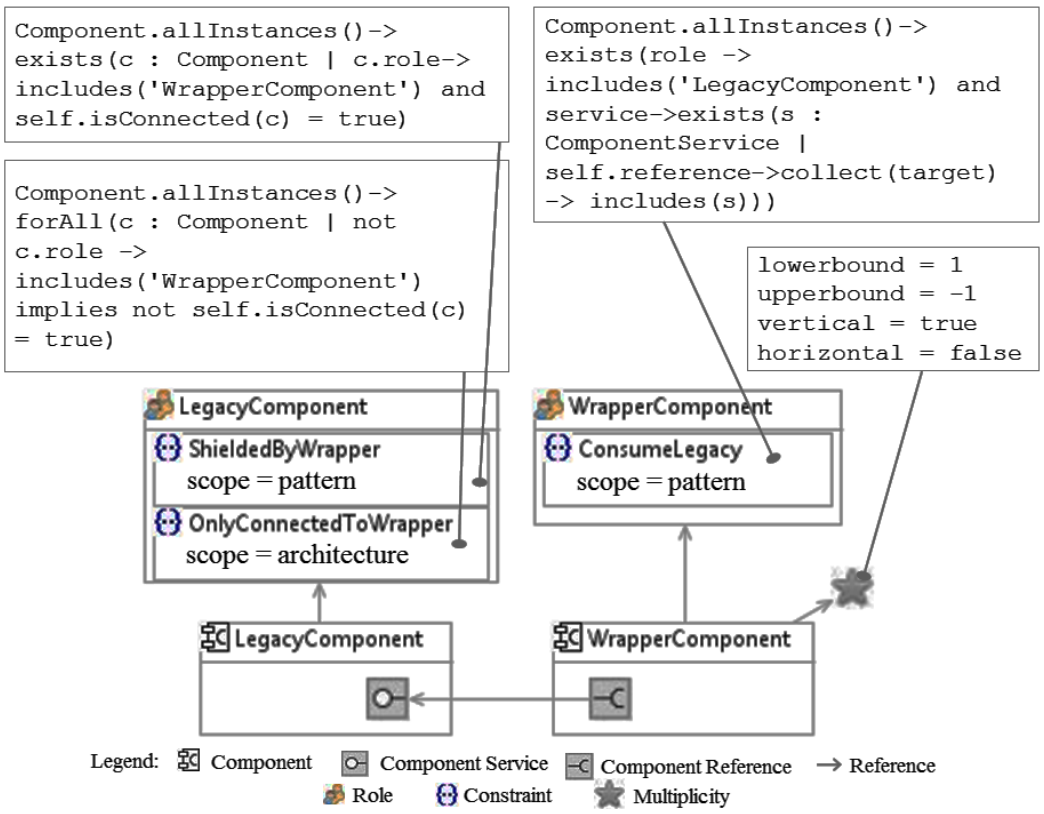

Figure 4.4: SOA Legacy Wrapper pattern model

Based on the *general pattern meta-model*, we can instantiate the *pattern model* for the SOA Legacy Wrapper with the emphasis on the following elements (as illustrated in Figure 4.4): the component *LegacyComponent* specified with the role *LegacyComponent* representing the component with legacy implementations and the component *WrapperComponent* specified with the role *WrapperComponent* representing the wrapper services in the pattern. The component *LegacyComponent* is not assigned with any multiplicity since it represents a concrete legacy component. Otherwise, the component *WrapperComponent* is assigned with a multiplicity

since it represents many possible wrapper components. Furthermore, its vertical multiplicity[2] indicates that there maybe many instances of WrapperComponent and they must be vertically connected. The role *LegacyComponent* is characterized by the *ShieldedByWrapper* constraint and the *OnlyConnectedToWrapper* constraint. The former stipulates that there must exist a component that plays the role *WrapperComponent* and is connected to the legacy component, the latter stipulates that for all existing components, if one does not play the role *WrapperComponent* then it can not be connected to the legacy component. Note that the *ShieldedByWrapper* constraint has a *pattern* scope since it involves only elements playing either the role *WrapperComponent* or *LegacyComponent*. The *OnlyConnectedToWrapper* constraint has an *architecture* scope since it involves not only elements playing the two mentioned roles but maybe also other elements in the architecture. The last constraint *ConsumeLegacy* characterizes the role *WrapperComponent* and stipulates that there must exist at least one legacy component with all services wrapped by the wrapper component. This constraint also has a pattern scope. Even though the other participating elements in the *Legacy Wrapper* pattern model, such as the component service of *LegacyComponent* and the component reference of *WrapperComponent*, do not have specific roles, they still contribute to the model to make a meaningful pattern.

4.3 Use of StAD

The process of using StADs consists of integrating pattern models to architectural models and checking the conformance of architectural models with associated StADs. The former is made thanks to a mapping model and the latter is made thanks to a particular view on the architecture.

4.3.1 Associating a Pattern to an Architectural Model

The association of a pattern with an architectural model consists in manually creating a mapping model between the former and the latter. Concretely, it links elements in the architecture that directly relates to elements from the pattern model. This is projected at the meta-model level as follows: each meta-class defines a mapping between the source, an architectural meta-class from the ADL meta-model and the target, an architectural meta-class from the *general pattern meta-model*. Figure 4.5 shows the mapping meta-model in case of SOA pattern. Elements from the left side of the figure are those from SCA meta-model. Elements from the right side of the figure are those from the SOA pattern meta-model. Each pair of these elements is linked by an mapping element from the mapping meta-model (in the middle of the figure) via the source and target references. All mapping elements are contained in the root mapping element.

The architecture of FRC case study [46] is chosen to illustrate the documentation of pattern use. Figure 4.6 sketches the mapping model which associates the Legacy Wrapper pattern to a

[2] upperbound = -1 indicates that there's no limited upper threshold for a multiplicity

Figure 4.5: SOA Mapping Meta-model

part of the FRC architecture. As we can see in this figure, two components *Data Controller* and *DWSA Data Service* in the FRC architecture are mapped respectively to two components playing the roles of *Legacy Component* and *Wrapper Component* in the *Legacy Wrapper* pattern model.

An architectural model can contain different mapping models, each of them represents a StAD made to the architecture. Thus, the architectural model can be considered as a set of elements in which each element can play different roles coming from the same StAD or different ones.

4.3.2 Filtering StAD views

The architectural views are useful for understanding the overall architecture of a complex system. In our case, each view represents one instantiation of a pattern in the architectural model. Thus, a view is produced by applying a transformation on the mapping model which captures a given StAD. The transformation serves as a filter to realize two purposes:

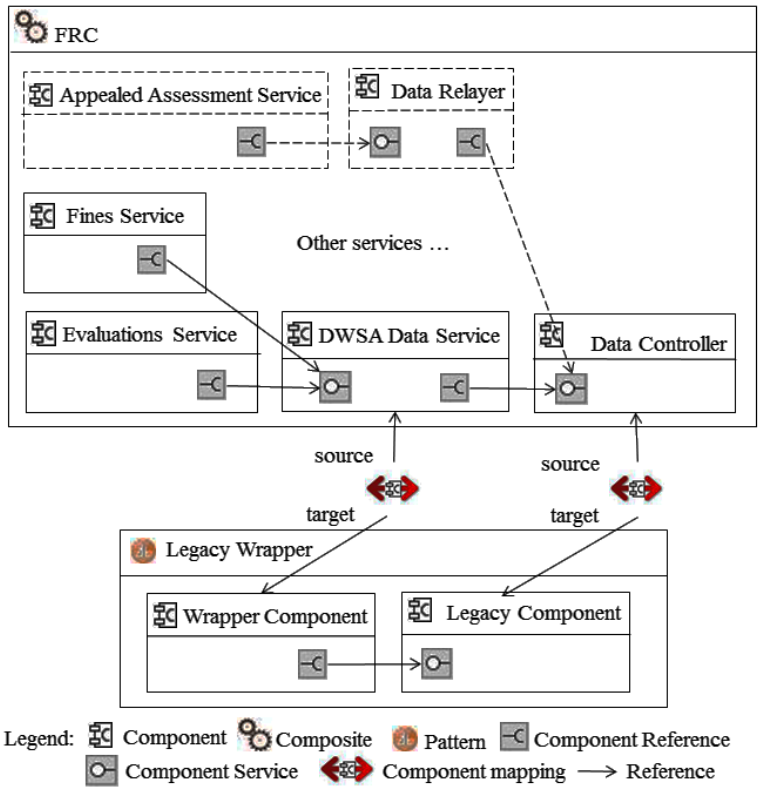

Figure 4.6: Mapping model for the *Legacy Wrapper* pattern in FRC (Elements in dashed line are those added after the evolution of the FRC architecture)

first, extract from the architectural model elements related to StADs and second, eliminate language-specific features of the architectural model to create a language-independent pattern view model. This transformation can be compared to the one from PSM (platform specific model) to PIM (platform independent model) in the MDA approach. To realize this, we leverage the MDA transformation techniques.

Algorithm 4 illustrates the transformation of a mapping model into a view model. First, it detects whether the pattern model has an *architecture-scope* constraint. If not, the algorithm creates a StAD view model that contains only pattern-related architectural elements. These elements and their pattern roles are determined via the mapping model. Otherwise, if the pattern model has at least one *architecture-scope* constraint, then the algorithm creates a StAD view model that contains all elements in the architectural model. But, only the roles related to the concerned pattern are kept in the elements. Patterns with *architecture-scope* constraints are not usual, as shown through our experimentation on well-known SOA patterns (see section 6.2.1).

The bottom of Figure 4.7 represents the filtered StAD view model for the mapping model before (without elements in dashed line) and after (with elements in dashed line) the evolution of the architecture described in Figure 4.6. As we can recall from the *Legacy Wrapper* pattern

Algorithm 4 The StAD view model filtering algorithm

Require: MappingModel MM
Ensure: StAdViewModel VM
1: PatternModel $PM \leftarrow MM.target$
2: ArchitecturalModel $AM \leftarrow MM.source$
3: Flag $f \leftarrow false$
4: **for all** Role $r \in PM.roles$ **do**
5: **for all** Constraint $c \in r.constraints$ **do**
6: **if** $c.scope = architecture$ **then**
7: $f \leftarrow true$
8: Break
9: **end if**
10: **end for**
11: **end for**
12: **if** $f = false$ **then** // No architecture-scope constraint is found // Only pattern-related elements are filtered
13: **for all** Mapping $m \in MM.mappings$ **do**
14: ArchitecturalElement $ae \leftarrow m.source$
15: PatternElement $pe \leftarrow m.target$
16: StAdViewElement $ade \leftarrow ae$
17: $ade.role \leftarrow pe.role$
18: $VM.add(ade)$
19: **end for**
20: **else** // At least one architecture-scope constraint is found // Filtering all elements
21: **for all** ArchitecturalElement $ae \in AM.elements$ **do**
22: StAdViewElement $ade \leftarrow ae$
23: **for all** Mapping $m \in MM.mappings$ **do**
24: **if** $m.source = ae$ **then**
25: $ade.role \leftarrow m.target.role$
26: **end if**
27: **end for**
28: $VM.add(ade)$
29: **end for**
30: **end if**

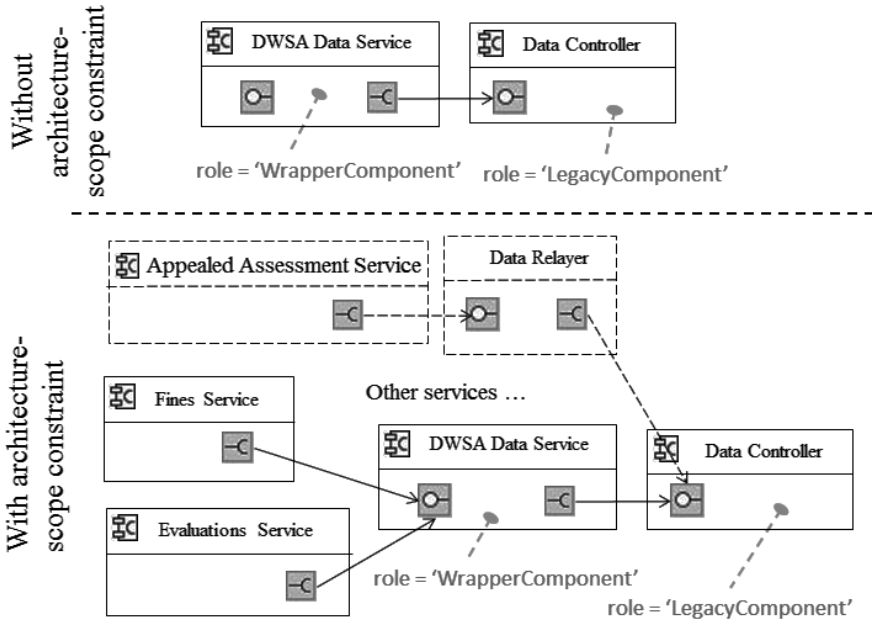

Figure 4.7: StAD views for the Legacy Wrapper pattern produced from FRC architecture (Elements in dashed line are those added after the evolution of the FRC architecture)

defined in Figure 4.4, the scope of the constraint *OnlyConnectedToWrapper* is *architecture*. This leads to the creation of a StAD view that contains all elements in the FRC architecture with only their roles related to the *Legacy Wrapper* pattern. In the produced StAD view there are two elements holding a role in the *LegacyWrapper* pattern: *DWSA Data Service* and *Data Controller* playing respectively the roles *WrapperComponent* and *LegacyComponent*. The choice of this illustrative pattern is made in order to cover the two kinds of constraint. In general cases architectural patterns hold mainly constraints with *pattern-scope*. To illustrate the StAD views generated in this case, we show on the top of Figure 4.7 the StAD view for the *Legacy Wrapper* pattern without taking into account its *architecture-scope* constraint.

4.3.3 StAD Checking

The checking process consists of two steps: i) The completeness of the mapping model is verified and if the mapping model's integrity is assured, ii) The second step, the StAD view meta-model is used to check the consistency of the StAD view model. Whenever the mapping model is detected as incomplete (e.g. due to the removal of some StAD-related elements in the architecture) or constraints imposed by the StAD view meta-models on StAD views are not satisfied, warnings are notified to the architect about which StAD is violated and which elements in the architectural model are involved.

The conformity of an architectural model with its associated StADs is checked through the

conformity of the corresponding *StAD view* with the concerned pattern model. For the purpose of checking, *StAD view meta-models* are generated from *pattern models*. The consistency of an *StAD view* is thus verified against its corresponding *StAD view meta-model* using the conformance operator from the MDA approach.

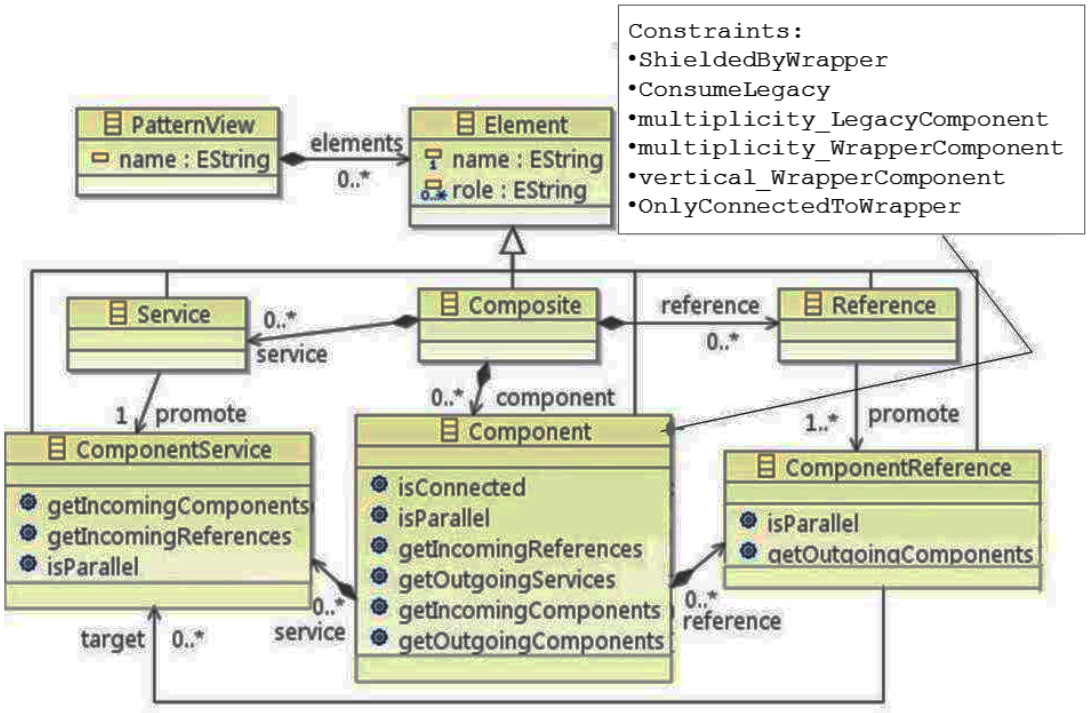

Figure 4.8: StAD view meta-model for the Legacy Wrapper pattern.

For every defined *pattern model*, an *StAD view meta-model* is generated containing meta-classes from the *general pattern meta-model* embedded with pattern constraints. The Algorithm 5 represents the simplified version of this transformation.

For instance, the Legacy Wrapper pattern model described in the previous section will be transformed to an *StAD view meta-model* with the participation of the following meta-classes: *Composite*, *Service*, *Reference*, *Component*, *ComponentService* and *ComponentReference* as shown in Figure 4.8. The StAD view meta-model is embedded with invariants imposed on the Component meta-class as follows:

```
//All legacy components must be wrapped with a wrapper
(1) invariant ShieldedByWrapper:
  if role->includes('LegacyComponent') then
    Component.allInstances()->exists(c: Component |
    c.role->includes('WrapperComponent') and
    self.isConnected(c) = true)
  endif;
```

Algorithm 5 The StAD view meta-model creation algorithm

Require: PatternModel M
Ensure: StAdViewMeta-model MM
1: $MM.add(createMetaClasses())$
2: **for all** PatternElement $pe \in M.elements$ **do**
3: Meta-class $mc \leftarrow MM.getCorrespondingMetaClass(pe)$ // Select the corresponding meta-class of the StAD view meta-model based on the pattern model element
4: **for all** Role $r \in pe.roles$ **do**
5: **for all** Constraint $c \in r.constraints$ **do**
6: $mc.add(c)$
7: **end for**
8: **if** $pe.getMultiplicity() \neq \emptyset$ **then**
9: Multiplicity $m \leftarrow pe.multiplicity$
10: $mc.add(createMultiplicityConstraint(m))$
11: **else** // Multiplicity constraint without parameter means that the meta-class should have exactly one instance
12: $mc.add(createMultiplicityConstraint())$
13: **end if**
14: **end for**
15: **end for**

```
//A wrapper must wrap at least one legacy component
(2) invariant ConsumeLegacy:
   if role->includes('WrapperComponent') then
      Component.allInstances()->exists(role->
      includes('LegacyComponent') and service->
      exists(s: ComponentService | self.reference->
      collect(target)->includes(s)))
   endif;

//Wrappers are vertically multiplied
(3) invariant vertical_WrapperComponent:
  if role->includes('WrapperComponent') then
     Component.allInstances()->forAll(role->
     includes('WrapperComponent') implies
     isParallel(self))
  endif;

//There is only one legacy component (single multiplicity)
(4) invariant multiplicity_LegacyComponent:
   let s: Integer = Component.allInstances()->
   select(role->includes('LegacyComponent'))->
   size() in s = 1;

//There maybe more than one wrapper (plural multiplicity)
(5) invariant multiplicity_WrapperComponent:
   let s: Integer = Component.allInstances()->
   select(role->includes('WrapperComponent'))->
   size() in s >= 1;
```

```
//There are no other component than wrappers can be connected to legacy component
(6) invariant OnlyConnectedToWrapper:
  if role->includes('LegacyComponent') then
    Component.allInstances()-> forAll(c: Component |
    not c.role -> includes('WrapperComponent')
    implies not self.isConnected(c) = true)
  endif;
```

A part of invariants on meta-classes correspond to constraints specified on role elements in the pattern model. The other part reflects information about orientation and multiplicity. We can observe through the example that the constraints imposed on the roles *LegacyComponent* and *WrapperComponent* in the pattern model are transformed into the invariants *ShieldedByWrapper* (1), *OnlyConnectedToWrapper* (6) and *ConsumeLegacy* (2) on the *Component* meta-class. The multiplicity of the role *WrapperComponent* in the pattern model is concretized in two other invariants in the *Component* meta-class: *multiplicityWrapperComponent* (5) and *verticalWrapperComponent* (3). Since the role *LegacyComponent* in the pattern model is not associated with any multiplicity, the invariant *multiplicityLegacyComponent* (4) restricts the exact number of *LegacyComponent* in the pattern view to one.

The FRC architecture passed through an evolution in which two components *Appealed Assessment Service* and *Data Relayer* (sketched in dashed line in Figure 4.6 and Figure 4.7) are added. As we can observe, its mapping model is complete. However, the addition of the *Data Relayer* component violates the *OnlyConnectedToWrapper* constraint (6) since it is directly connected to a component playing the role *LegacyComponent*.

Similar to UML, this way of StAD specification allows us to introduce two levels of consistency: meta-model and well-formedness rules. More specifically, well-formedness rules are expressed in OCL to assert the syntactic correctness of StAD view models. According to the classification of model consistency methods presented in [29], this approach is a syntactic-horizontal consistency one.

4.4 Summary

In this chapter we presented a solution to the automatic checking of StAD by combining pattern model and mappings. We first introduced a general process in which patterns are leveraged to document StAD. By using our existing pattern description language, patterns are modelled once and later used to document StADs. ADs are made by mapping a predefined pattern model to the architectural model. StAD violation is then checked against the architectural model using both pattern model and mappings. Once violation is detected, the architect can choose to modify the architectural model to avoid the violation or go back to the definition of AD. We do not rewind back to the pattern definition part which is already presented in Chapter 3. However, since we illustrate the approach on SOA, we presented basic concepts from

SCA, accompanied with the example of the SOA *Legacy Wrapper* pattern. Via the mapping meta-model, we explained how to map pattern elements to architectural elements each time we want to make an AD. The benefit of mappings is twofold: i) From the mapping model, we can obtain a filtered view of AD-related element, ii) The information about the existence of AD-related elements can be stored in the mapping model. We showed the algorithm to filter a StAD view from the architectural model and an illustrative example of an architecture applied with the *Legacy Wrapper* pattern. This step aims to provide a reduced architectural view which contains only StAD-related elements. Finally, we demonstrated the StAD violation checking method which involves both the pattern model and the mappings. While the pattern model helps maintain the structural consistency of the StAD, the mappings on the other hand help verify the existence of StAD-related elements.

The idea of using pattern and mappings as a means to verify StAD violation has been presented in the WICSA/ECSA conference [42] and the Automated Software Engineering journal [45].

5 Implementation

Contents

Both ideas of a composition-supported pattern description language presented (Chapter 3) and a StAD documentation approach (Chapter 4) are put into practice via two tools: ADManager and COMLAN respectively. In this chapter ADManager and COMLAN are presented through a set of use cases and how they are realized using these tools.

5.1 COMLAN tool

COMLAN is a tool to design architectural patterns with the focus on documenting composition operations. In COMLAN tool, merging operators are used as model element. Specifically, one can store merging operators in a persistent form, reference to merging operator from another element, etc. The following presents supported use cases in COMLAN and its architecture.

5.1.1 Use cases

The use cases present a wide area of issues concerning the problems discussed in the introduction of COMLAN in Chapter 3. COMLAN tool responds to the following use cases:

1. Create architectural patterns

   Use case: Given an architectural pattern description, create a coherent pattern model with all the necessary concepts.

   COMLAN: The tool realizes the concrete syntax of the language via a graphical representation. One could select necessary pattern elements from a design panel and modelize his pattern. The created pattern model can be verified to be validated or not with respect to its abstract syntax defined in the meta-model.

2. Compose patterns using merging operators

   Use case: Use merging operators as first-class entity to combine patterns.

   COMLAN: Merging operators also appear in the design panel together with other pattern elements. One could drag and drop two different unit pattern models, add an appropriate merging operator and link it to corresponding elements in each pattern.

3. Refine the composed pattern

   Use case: Refine the composed pattern by concretizing merging operators.

   COMLAN: In the matter of a click, a refined version of a composed pattern model is automatically produced. Also, the refined pattern model is syntactically checked against the meta-model to verify the correctness of the transformation.

4. Trace back to constituent patterns

   Use case: From a given pattern model, trace back to the constituent pattern models (if they exist)

   COMLAN: Any composed pattern element is tagged with the merging operator from which it is originally created. Via the containing model of this merging operator, one can obtain the general picture of pattern composition.

5. Reconstruct the pattern via composition

   Use case: From a given pattern model, changes in constituent pattern models (if they exist) should be reflected.

   COMLAN: Any change in a given pattern model can be propagated in the composed pattern in which it participates. From the changed version of the composed pattern, various pattern variants can be obtained.

5.1.2 COMLAN architecture

COMLAN is based on EMF (Eclipse Modelling Framework)[1]. We chose EMF to realize our tool since we leverage MDA, where models are basic building units, to develop our approach. Figure 5.3 depicts the architecture of the COMLAN tool.

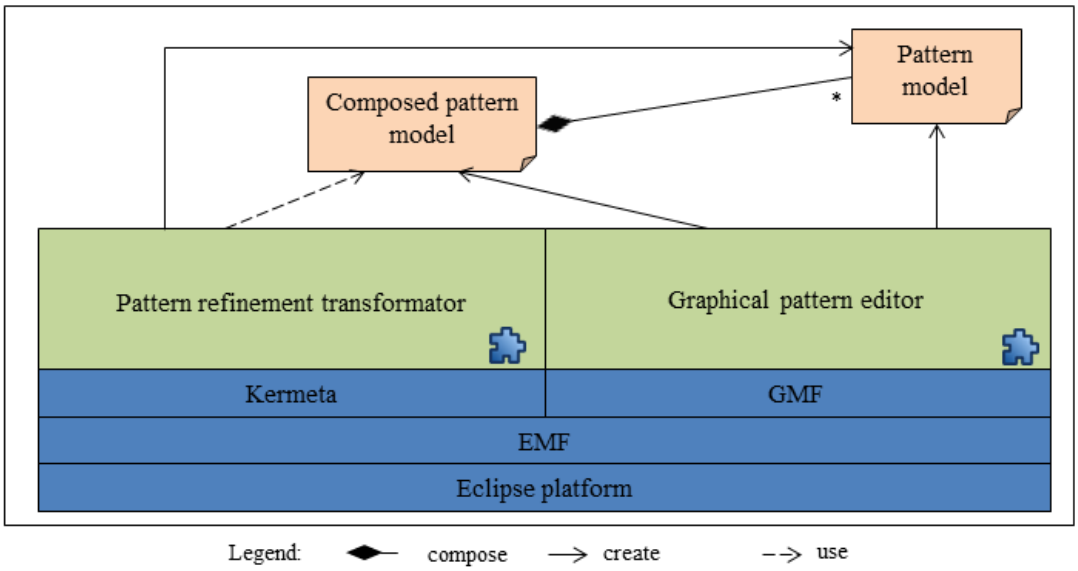

Figure 5.1: COMLAN tool architecture

The tool consists of two Eclipse plug-ins built on existing Eclipse technologies:

- *Pattern editor plug-in* uses EMF and GMF (Graphical Modeling Framework)[2] modeling facilities in order to allow architects to define *Pattern models* graphically. Two types of pattern models are supported using the graphical pattern editor: unit pattern models and composed pattern models. Composed pattern models are designed by selecting patterns from a catalogue and composing them using two types of merging operators: stringing and overlapping. Hierarchical pattern description is also supported via the inclusion

[1] More details about EMF are accessible at: http://www.eclipse.org/modeling/emf/"
[2] More details about GMF are accessible at: http://www.eclipse.org/modeling/gmp/"

of an entire pattern inside a pattern element. Besides, the editor allows the automatic propagation of changes in the constituent patterns to the composed pattern in which they participate. Figure 5.2 represents several snapshots of COMLAN tool. The bottom-left shows the graphical pattern editor. It contains the example of two constituent patterns *ConnectedLayer* and *StrictOrder* which are combined using an overlapping operator between the *Layer* component and the *OrderedComponent*. The bottom-right shows the panel from which pattern elements can be chosen. The top-right depicts the property window for the Layer component. It has a multiplicity and plays the role of a Layer. Finally, the top-left shows the context menu where users can perform the pattern composition functionality.

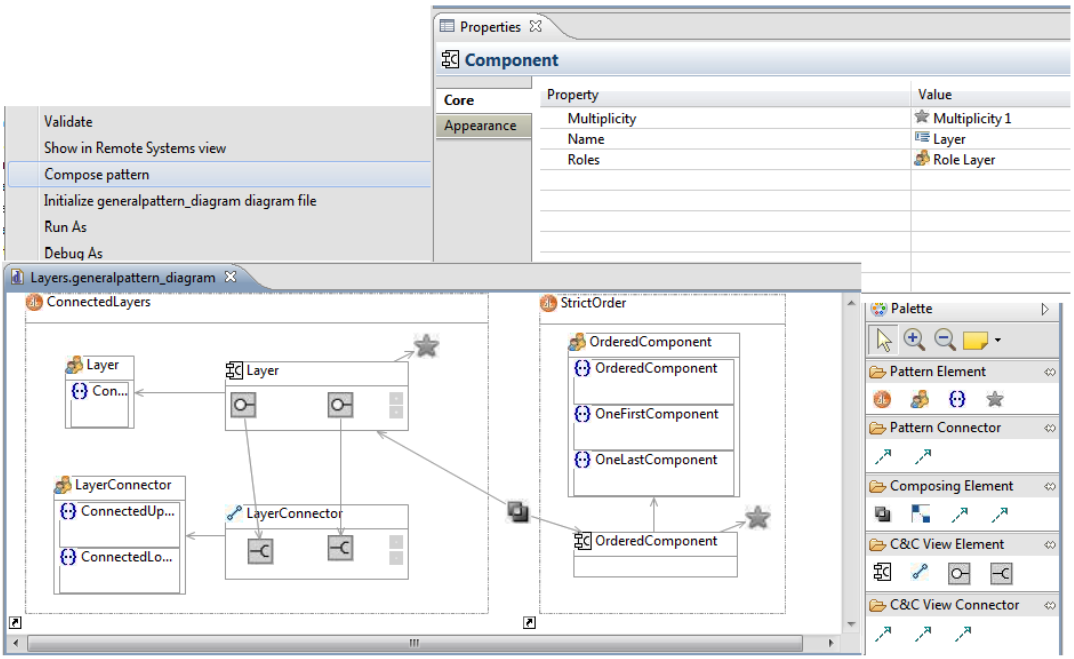

Figure 5.2: Snapshots of COMLAN tool

- *Pattern refinement plug-in* uses *Kermeta* [3] to implement rules transforming composed pattern model to refined pattern model. The plug-in takes as input composed pattern models obtained from the pattern editor and produces as output the refined models. This functionality allows the architect to obtain a pattern with all the merging operators concretized. The refined pattern model is then accessible using the pattern editor, allowing it to participate in further pattern compositions.

The reader is invited to visit the COMLAN website [4] for a complete tutorial and a video about this tool and the example of Vistrails's architecture [8] pattern modelling.

[3] Kermeta is described in details in [34]

[4] http://www-archware.irisa.fr/software/comlan/

5.2 ADManager tool

ADManager is a tool to document StAD using pattern model. In ADManager tool, one can define a pattern model, create an AD by mapping the pattern model to the architectural model and verify the conformance of the architectural model against the created StAD after a change in the architecture. The following presents supported use cases in ADManager and its architecture.

5.2.1 Use cases

The use cases presented in this section are drawn from the ideas shown in Chapter 4. As a StAD documentation tool, ADManager responds to the following use cases:

1. Create architectural patterns

   Use case: Given an architectural pattern description, create a coherent pattern model with all the necessary concepts.

   ADManager: ADManager reuses COMLAN as its architectural pattern model creator. Thus, any architectural pattern and its composition can be realized in the same way as COMLAN does. Additionally, SOA patterns are also supported.

2. Integrate StADs to architectural models

   Use case: As we know, StADs are represented by pattern model. Thus, this use case concerns with how to relate a pattern model with an architectural model.

   ADManager: Pattern models are mapped to architectural models using mapping elements. One can create a mapping model, select a corresponding pattern model and an architectural model and then map elements from the former to those from the latter.

3. Extract pattern-view

   Use case: An architectural model gets involved in different StAD models which leads to the complexity of comprehension and violation detection. Thus, for each applied StAD one should be able to extract only the related pattern elements.

   ADManager: From the mapping model, one could extract a pattern-view model which consists of only the StAD-related elements. All of the other elements are filtered out.

As *ADManager* is also a consistency management tool, it supports the following functionalities among those described in [14]:

- Automatically detect inconsistency

  Use case: At any moment, one should be able to detect if changes made to the architecture lead to possible conflict with create StADs.

ADManager: The tool automatically extracts pattern view models from mapping models and checks their consistency.

- Present inconsistency

  Use case:

  ADManager: The tool provides a visual inconsistency feedback to the user via description dialogues. More specifically, a summarized report about which StAD is violated and if so, at which constraint.

5.2.2 ADManager architecture

In its actual version, ADManager supports the documentation of StADs in three different languages: SCA [6], Acme [17] and PiADL [37].

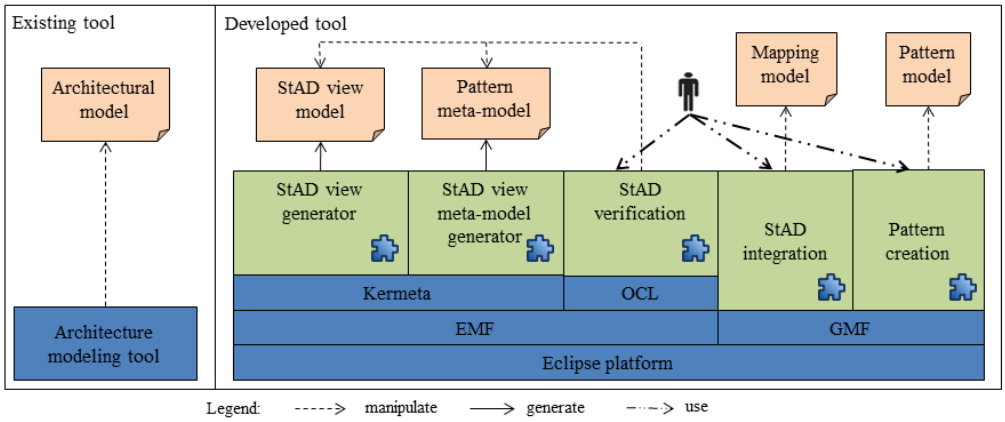

Figure 5.3: The architecture of ADManager

ADManager is developed based on EMF (Eclipse Modelling Framework) [40]. As shown in Figure 5.3, the tool consists of five Eclipse plug-ins built on existing Eclipse technologies. They are:

- *Pattern creation plug-in* uses EMF and GMF (Graphical Modeling Framework)[5] modeling support in order to allow architects to define *Pattern models* graphically.
- *StAD integration plug-in* is an editor supporting the creation of *Mapping models* between pattern elements and architectural model elements.
- *StAD verification plug-in* uses OCL tool to support writing rules in pattern models, during pattern creation, as well as conformance verification between StAD view models and StAD view meta-models during StAD checking.

[5] http://www.eclipse.org/modeling/gmp/

- *StAD view meta-model generator plug-in* uses Kermeta to implement rules generating StAD view meta-models from pattern models.
- *StAD view generator plug-in* uses Kermeta to implement rules generating StAD views from mapping models.

Note that in Figure 4.3 the *General Pattern Meta-Model* for SOA is separated into two parts: one specific to the SOA description language family and the other for the notion of pattern. The separation of these two aspects gives our tool the flexibility to support many different ADL families just by switching to the appropriate structural part of the pattern meta-model. Indeed, besides SCA, we have been able to support two different ADLs, namely Acme and PiADL, by keeping the pattern part and modifying the structure part in the pattern meta-model[6].

Figure 5.4 represents a snapshot of ADManager tool. This is actually the mapping model aiming to link the Layer pattern model on the left to an architectural model on the right. As we can observe, each mapping element has a reference pointing to a role-playing element in the pattern model and another reference pointing to a respective element in the architectural model.

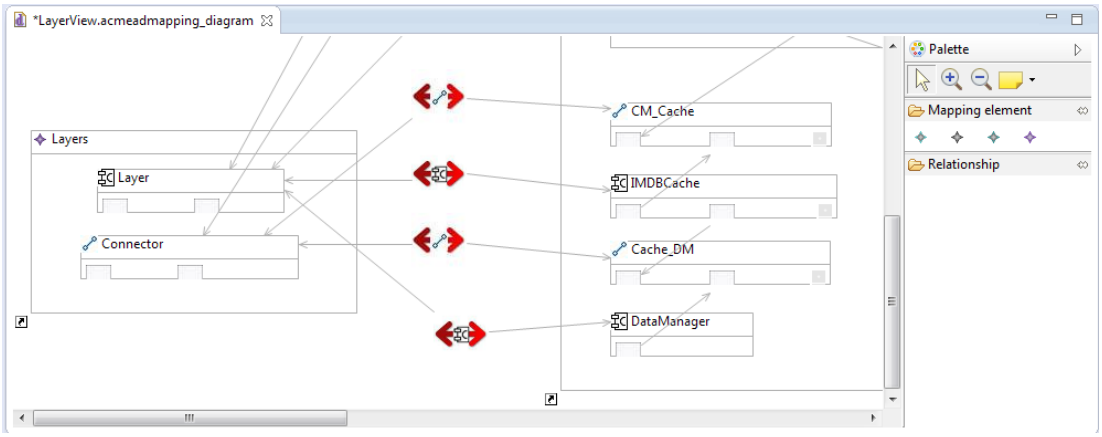

Figure 5.4: Snapshots of ADManager tool

The reader may obtain a complete guiding tutorial video and more information about the *ADManager* tool at: http://www-archware.irisa.fr/software/admanager/.

5.3 Summary

This chapter presented two tools ADManager and COMLAN corresponding to two mainstream ideas of the thesis. While COMLAN describes itself as a tool to model patterns, ADManager is used to document StADs and verify their consistency. They are both built based

[6]The reader can find these meta-models at the same website of ADManager tool

on EMF and GMF which favour the MDA approach. These technologies center themselves around the notion of model. Two important benefits from model-based approaches lies in : i) The built-in meta-model/model conformance verification support and ii) The transformation mechanism. Every concept in a language is described using model and the transition from one model to another is realized via model transformations. In COMLAN, the pattern/pattern instance conformance is assured via the meta-model/model relation and model transformation is used to refine merging operator. In ADManager, the StAD/StAD-related model element conformance is assured via the meta-model/model relation and model transformation is used to filter out StAD-related elements. For each of these two tools, we showed its architecture and its supported use-cases. We also explained how each use-case responded to the initial requirements for each approach. Some screen shots and links to tutorial videos are also provided.

6 Empirical evaluation

Contents

To evaluate the two approaches that are respectively presented in chapter 3 and chapter 4, we separately conducted two empirical experiments. The first experiment consists in applying the approach on the composition of a set of formalized architectural patterns, including their variants to show that composed patterns have become traceable and reconstructable. The second experiment applies the approach on a set of architectural models to show that architectural decisions are well explained and all of their violations are detected. In this chapter we present these two empirical evaluations in detail.

6.1 Empirical evaluation for pattern composition approach

Our approach focuses on giving pattern merging operators first-class status to support the traceability and the reconstructability of patterns. Thus, the approach is evaluated on the interest of using merging operators in: i) tracing back constituent patterns in pattern composition. ii) reconstructing composed patterns.

6.1.1 Experimental setup

The materials used in our experiment are patterns we gather from different sources of architectural patterns in the literature such as [2, 9, 53, 11, 46]. We distinguished two levels of pattern granularity: primitive level and architectural level. As being shown via the Acyclic pattern in the illustrative example (Section 2.5.2.1), we also consider the common structures used in patterns as primitive patterns. In existing work, these structures are described by different terminologies such as architectural constraints in [9, 11] or architectural primitives in [53, 54]. However, considering the ability to combine these structures to build patterns, they are also treated as patterns at the primitive level in our approach. At the architectural level, patterns are modeled using the information in the structure part of the pattern description. Indeed, the structure description of the pattern is an important source to detect whether it is possible to construct the pattern by composing other patterns using overlapping or stringing operator. In total, 16 architectural pattern definitions are used in our study. They cover patterns in different categories and viewpoints, from data flow, data-centering to distribution, etc. Taking the variability of patterns into consideration, a given pattern can exist in different variants. Except for the pure variant, which represents the characteristics of the pattern as is, the more relaxed variants of patterns also integrate the structure of other patterns to adapt to different needs. For instance, one of the variants of the *Pipes and Filters* pattern is the *Layered Pipes and Filters* pattern. It is slightly different from the pure form of *Pipes and Filters* with Filters structured in layers. From 16 collected architectural pattern definitions, we could find 28 variants. In average, there are 1.75 variants per pattern definition. Table 6.1 shows the catalogue of pattern definitions and their variants. The complete catalogue of patterns and variants used in our experiment can also be found at Appendix A.

We do not evaluate the correctness of the traceability and the reconstructability of pattern composition in our approach. The reason for that is twofold. First, it is quite obvious that by switching from pattern designed in step 2 (refined pattern) to step 1 (composed pattern), we should be able to detect which patterns are used to form the composed pattern. Second, the ability to reconstruct pattern from merging operator is ensured by the correctness of our transformation algorithm. However, we empirically evaluated how much necessary it is to trace back to constituent patterns and to reconstruct patterns. Thus, we have two hypotheses to validate.

Hypothesis 1 Most of pattern structures can be decomposed to other fine-grained pattern struc-

Table 6.1: Pattern catalogue

Pattern definitions	Pattern variants
P1-Enabled cycle	V1.1-Enabled cycle [2, 9]
P2-Forbidden cycle	V2.1-Forbidden cycle [2]
P3-Shield	V3.1-Shield [53]
P4-Layers	V4.1-Basic Layers [2, 9] V4.2-By-passed Layers [2] V4.3-Not By-passed Layers [2, 9] V4.4-Client-Server Layers [2] V4.5-Filtered Layers [2]
P5-Pipes & Filters	V5.1-Basic Pipes and Filters [2, 9] V5.2-By-passed Pipes and Filters [2] V5.3-Pipeline [2, 9] V5.4-Layer-structured Pipes and Filters [2] V5.5-Data sharing Pipes and Filters [2]
P6- Shared Repository	V6.1-Basic Shared Repository [2, 9, 11] V6.2-Layer-structured Shared Repository [2]
P7-Microkernel	V7.1-Basic Microkernel [2] V7.2-Microkernel with Broker [2]
P8-PAC	V8.1-PAC [2, 9]
P9-Indirection Layer	V9.1-Indirection Layer [2]
P10-Client-Server	V10.1-Basic Client-Server [2, 9, 11] V10.2-Client-Server with Broker [2] V10.3-Client-Server with Microkernel [2]
P11-MVC	V11.1-MVC [2, 9]
P12-Proxy	V12.1-Proxy [2, 9]
P13-Broker	V13.1-Broker [2, 9]
P14-Façade	V14.1-Façade [53, 46]
P15-Legacy Wrapper	V15.1-Legacy Wrapper [11]
P16-Data-centered Pipes and Filters	V16.1-Data-centered Pipes and Filters [11]

tures

Hypothesis 2 Most of composed patterns can be reconstructed from other patterns by adapting their constituent patterns

The next two sections aim at validating these two hypotheses.

6.1.2 Traceability

We first counted the number of pattern variants from which the structure can be deduced by merging other patterns. The counting process is semi-automatic. First, beginning with pattern variant descriptions from different sources we can form a graph of pattern relationship. Each node of this graph represents a pattern and an arc between two nodes represents the composition relation between two patterns. Next, the graph is used to count pattern variants and the frequency of composition. Figure 6.1 shows that a large number of pattern variants (19 over 28) can be composed from other variants. The number of constituent patterns equals zero means that the variant is at the primitive level or it is a monolithic pattern. An example of this could be the Shield architectural primitive (V3.1) which is a fundamental modeling element to build more complex pattern. A variant which is composed from only another pattern represents the situation in which one or several elements of the pattern are structured by combining with another pattern. For instance, a variant of the *Pipes and Filters* pattern is the case where the *Filter* is internally structured by a *Layers* pattern (V5.4). This variant is in fact formed by the combination of the *Filter* component and the entire *Layers* pattern. Finally, being composed from two other patterns means that the pattern encompasses the two constituent patterns taking into consideration overlapped elements. For example, in the *Data-sharing Pipes and Filters* variant (V5.5), the *Pipes and Filters* pattern (V5.1) is combined with the *Shared Repository* pattern (V6.1) by overlapping *Filter* components and Data accessor components. As we can observe, 19 over 28 variants, which is equivalent to 67.86% of the variants, can be composed from at least another pattern. This partially explains the need of tracing back constituent patterns for a given pattern variant.

We then evaluated the frequency of using a given pattern to compose the other ones. Therefore, another question about the traceability is what is the probability to trace back to the same pattern in different cases of pattern composition? To address this issue, we counted all cases of pattern composition that can be performed by using a variant. Figure 6.2 shows that 11 over 28 pattern variants, which is equivalent to 39.29%, can be used in at least one composition of pattern. Especially, two pattern variants of the Pipes and Filters pattern and the Layers pattern (V1.4.1 and V1.5.1) are used in six compositions of other pattern variants. For the latter, this is explained by the fact that the Layers pattern is often used to construct the internal structure of other patterns. Similarly, the Pipes and Filters pattern is often integrated with other patterns to form different variants such as Pipeline (V5.3), Data-sharing Pipes and Filters (V5.5), ect.

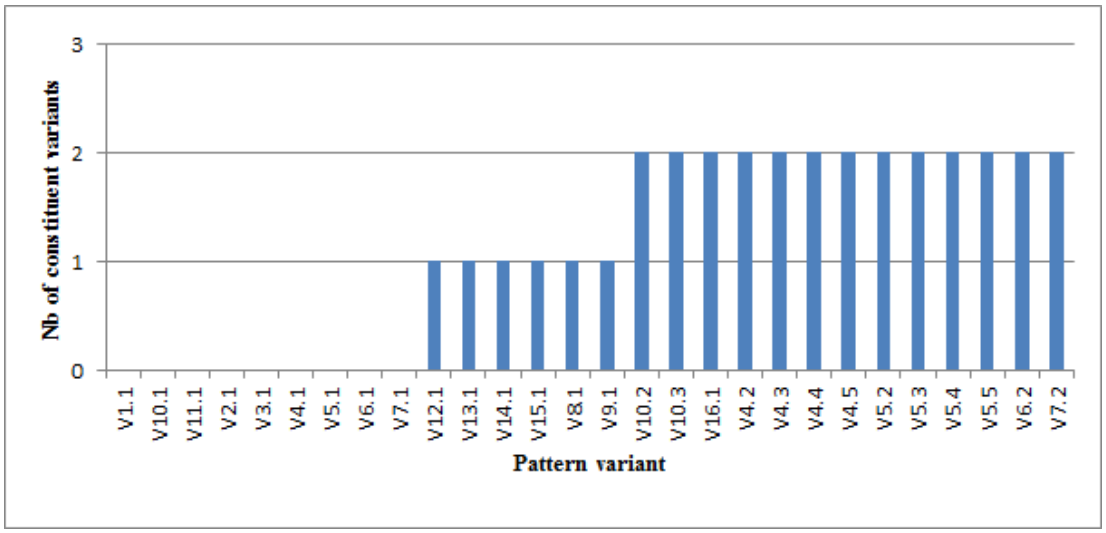

Figure 6.1: Most of pattern variants can be composed from other variants

In average, a given pattern variant can be found in 1.14 compositions of patterns. Thus, this reinforces our hypothesis on the need to trace back the constituent patterns.

6.1.3 Reconstructability

In our approach, reconstructability is defined as the ability to create another pattern from an existing one just by reusing a part of it and its merging operators. We found that this phenomenon often occurs in the composition with different variants of the same pattern. Pattern variants share the characteristics of the pattern definition, only a part of the structure differs from one to another. Thus, reconstructing a composed pattern boils down to keeping one constituent pattern structure, replacing the variant-related structure by another appropriate one and reapplying the merging operators. An example of the reconstruction is the variant V5.4 of the Pipes and Filters pattern where the Filters are internally structured by the Layers pattern. This variant is in fact the composition of the Pipes and Filters pattern and the Layers pattern. There exist totally five variants of the Layers pattern which leads to the possibility to have five composed patterns. Reconstructing a composed pattern from another one is in fact the matter of replacing different variants of the Layers pattern during the composition process as shown in Figure 6.3. Indeed, the Layers pattern exists in different variants and switching among them during the composition produces different composed pattern variants.

Taking this remark into account, in order to evaluate the interest to have the pattern reconstruction possibility, for each pattern variant we applied the approach and measured how many other variants can be built from it. Figure 6.4 shows for each composed pattern, the number of possible reconstructions.

The variants chosen to participate in this evaluation are those created from the composition

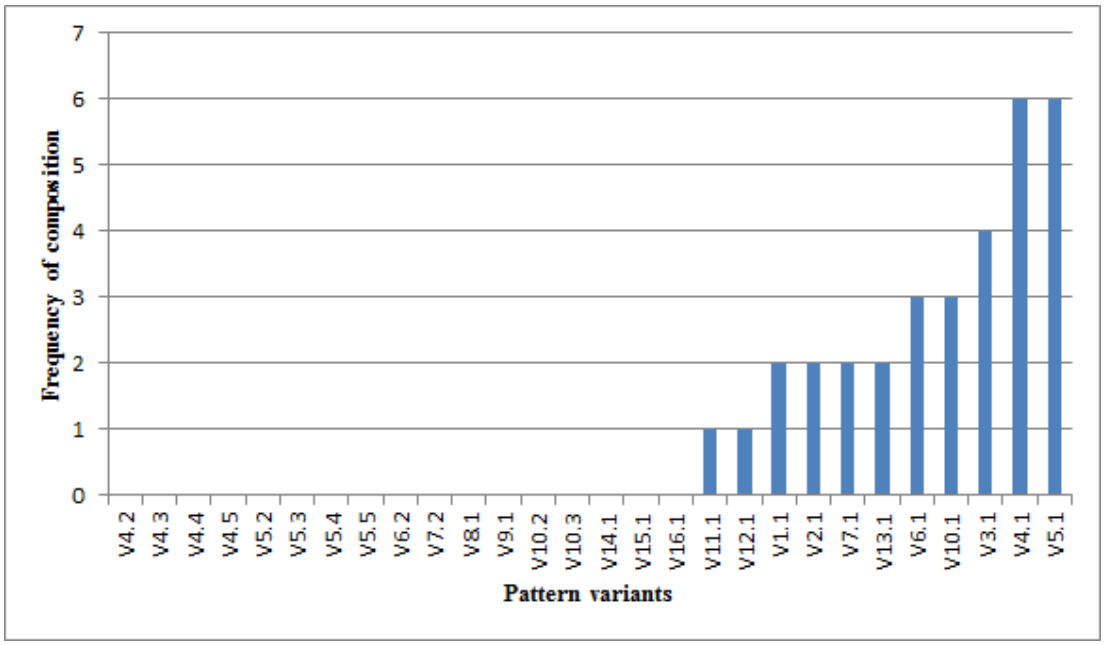

Figure 6.2: Frequency of composing a pattern variant

of at least two patterns which reduces the dataset to 13 variants. The primitive variants are excluded since they are not the products of any composition process. As we can observe from Figure 6.4, 54% of the chosen variants (7 out of 13 variants), involve in at least two reconstructions. In particular, the variant Data-centered Pipeline (V5.1), which is the composition of the Pipes and Filters pattern and the Shared Repository pattern, involves in 10 reconstructions. This is explained by the fact that there exist 5 variants of Pipes and Filters and 2 variants of Shared Repository. Thus, 10 possible compositions can be made by switching Pipes and Filters variants and Shared Repository variants respectively. Thus, this result shows that the reconstruction may concern a reasonably large cases of pattern composition.

6.1.4 Discussion

In our study we do not consider the combination of variants from the same pattern definition. The combined variant, if existed, would capture the characteristics of constituent variants. For instance, there may exist a combination of the *Layer-Structured Pipes and Filters* pattern (V5.4) and the *Pipeline* pattern (V5.3). The combined pattern would be a Pipes and Filters pattern that does not allow cycles among Filter components and all the Filter components are internally structured by the Layers pattern. Despite of the feasibility of this kind of combination, we have not been able to find any related work mentioning it. Thus, the combinations of variants of the same pattern definition were excluded from our study.

We only studied the reconstruction of pattern variants of the same pattern definition. How-

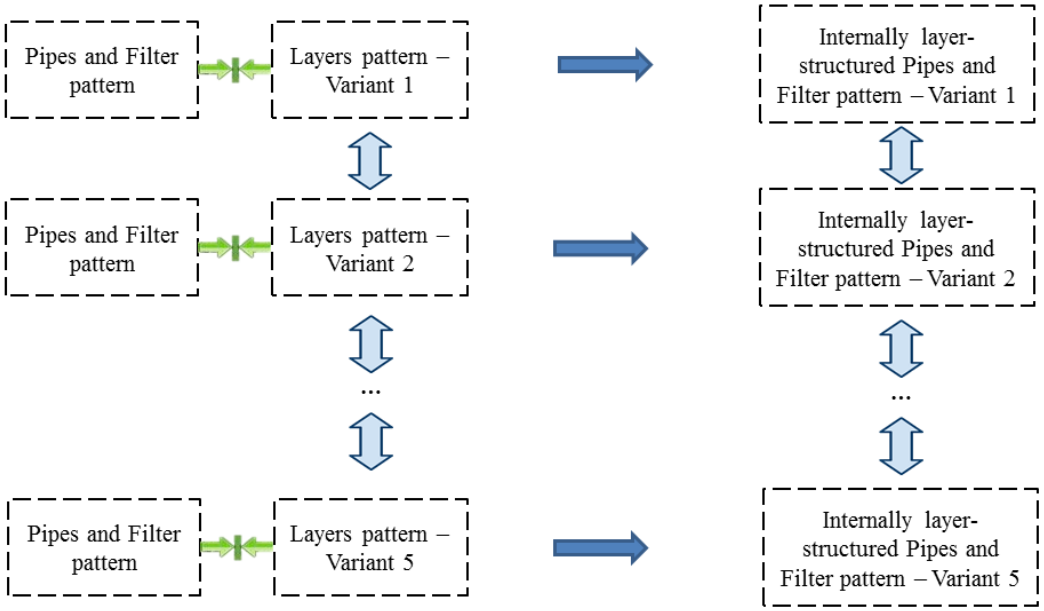

Figure 6.3: Reconstructability of composed pattern by switching between different variants of constituent patterns

ever, we do not exclude the ability to reconstruct a variant of a pattern definition using a variant from a different pattern definition. This situation does not exist within the scope of the architectural patterns collected in our study. Nevertheless, it may exist in an extended dataset using a broadened library of patterns.

6.1.5 Threats to validity

Our study is concerned by internal and external threats to validity.

Internal validity: The determination of pattern composition could be biased by the fact that the researchers participating in the pattern composition detection process already know about the pattern composition operators. Moreover, architectural primitives or unit patterns are sometimes implicitly described in patterns' specification and we risk having some of them undiscovered. We mitigated these risks by having pattern compositions discovered by different members and making sure that pattern composition specifications are drawn from many different sources. Thus, the correctness of our catalogue about pattern composition is assured.

External validity: All the architectural patterns used in our study and their variants are mainly collected from existing work in the literature. Although a wide spectrum of architectural patterns is covered, the study cannot generalize the effect of our approach in the support of pattern composition in general, considering the flexibility and customizability of patterns [9]. Indeed, many patterns capture existing experience that are specific to certain projects, software

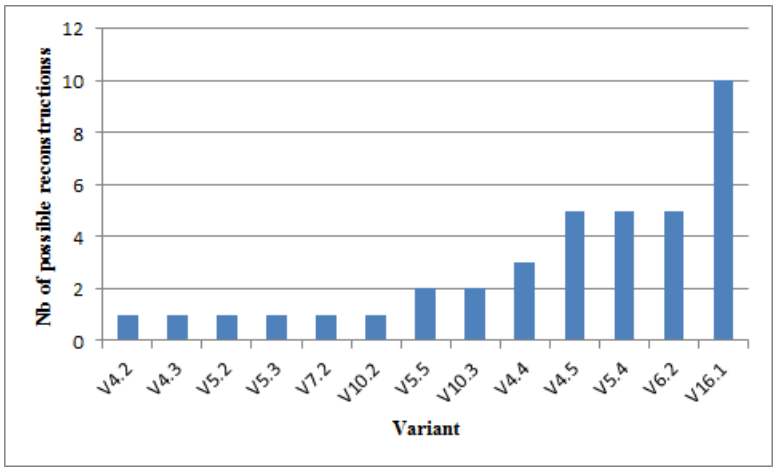

Figure 6.4: Frequency of composed pattern reconstruction by reusing merging operators

systems or companies. However, the more patterns, the more crowded the variants and thus the more likely the approach has effect.

6.2 Empirical evaluation for StAD documentation approach

The main contribution which lies behind our work is the documentation of pattern-centric StADs which maintains both the existence of StADs and their structural consistency. For this purpose we defined a general pattern definition language that can be switched from one paradigm to another. Thus, to evaluate our approach, we first show the expressiveness of our pattern definition language in two different paradigms. Then we show the effectiveness of StADs'documentation and their completeness in terms of existence and structural consistency.

6.2.1 Application of pattern definition language

To evaluate the support of multi-paradigms, we collected patterns from two different paradigms, namely SOA and Component-Based Architecture (CBA), and see how our pattern definition language can support them. There are two criteria upon which patterns are chosen to be formalized. The first one is that the pattern's vocabulary cannot extend beyond the concepts supported by the corresponding ADL. The second one concerns the scope of the pattern. We limit selected patterns to the structural aspect, other patterns are considered to be out of scope.

6.2.1.1 SOA patterns

We have examined the SOA patterns from [46] that are summarized in table 6.2. In that table we reused the categorization of patterns given in [46]. Among the 80 identified patterns there are up to 50 patterns focusing on the aspect of service management. Examples of service

management patterns are those concern how to physically centralize or decentralize services, how to determine the boundary of service logic, etc. These patterns in fact do not directly concern the structural aspect of the architecture. Therefore, they cannot be formalized using concepts from the ADL.

Table 6.2: Categories of SOA Patterns from [46]

Pattern category	Patterns	Architectural patterns	Formalized patterns	Patterns with architecture scope constraints
Service inventory design patterns	24	10	5	0
Service design patterns	35	16	16	2
Service composition design patterns	23	6	6	2
Total	82	32	27	4

As we can observe in the Table 6.2, among the remaining 32 architectural patterns there are ones based on architectural concepts that are not supported yet by Service-Oriented ADLs such as service inventory, service layer, etc. This explains why only 27 patterns are formalized using our approach (“Formalized patterns” column). Most of the formalizable patterns fall into the Service design pattern category. Indeed, patterns in this category are good practices in service organization, encapsulation, implementation, governance,Ä¶ and therefore, suitable to be architecturally formalized. The column “Patterns with architecture scope constraints” gives the number of patterns holding at least one constraint with architecture-scope. Only 4 patterns, among the 27 formalizable ones, fall in this case.

6.2.1.2 CBA patterns

To support the design of CBA patterns, starting from the SOA pattern meta-model (see Section 4.3), we switched the structural part to another one that conforms to CBA patterns. Figure 6.5 shows the CBA pattern meta-model. More specifically, this structural part consists of the set of architectural elements: component, connector, port and role. This set of elements is in fact the necessary design vocabulary for architectural pattern as pointed out in [33, 11]. Switching from the SOA pattern meta-model to the CBA pattern meta-model is in fact the matter of disconnecting the structural part of the former and connecting the structural part of the latter. More specifically, connecting the CBA meta-model to the pattern part consists in i)

Adding the inheritance relationship between two meta-classes SimpleElement and CompositeElement (CBA part) and the Element meta-class (Pattern part) and ii) Adding a composition relationship between the CompositeElement meta-class (CBA part) and the Element meta-class (Pattern part). Thus, the switching is realized without any modification in the pattern part of the general pattern meta-model.

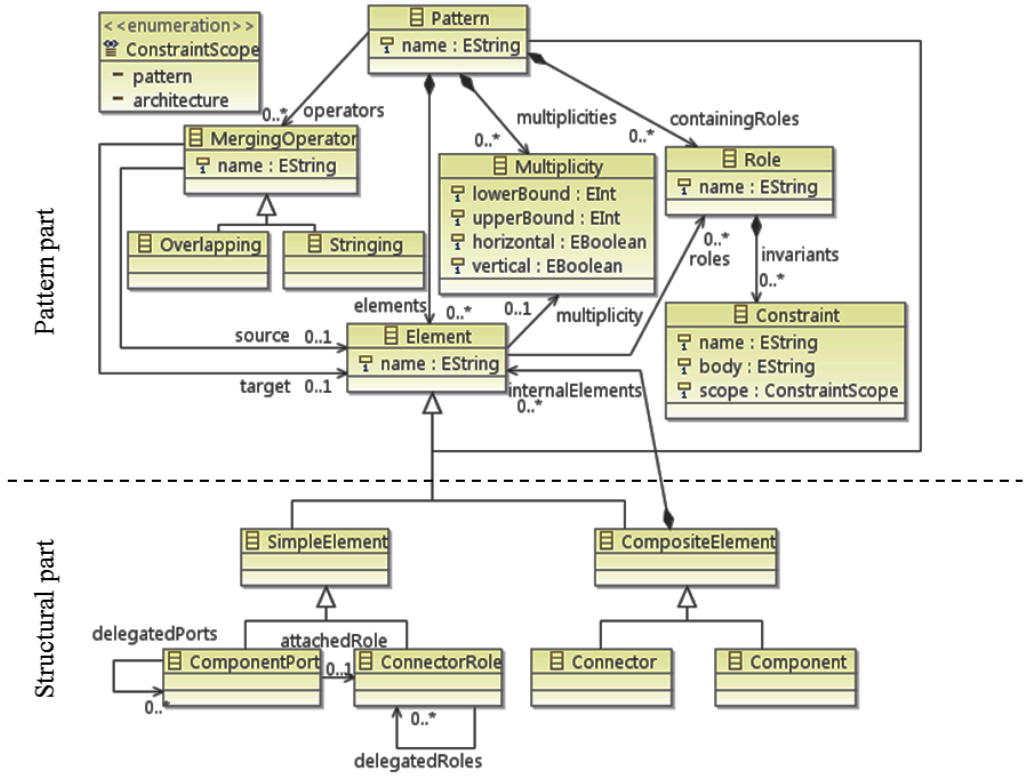

Figure 6.5: General CBA pattern meta-model

Using this adapted pattern language, we have tried to model the CBA pattern catalogue described in [2]. Table 6.3 shows the examined patterns assigned to different viewpoints.

As we can observe, we have been able to model 14 patterns (Column Structural patterns) out of the total 24 patterns (Column Pattern). Our pattern definition language focuses on the structural aspect of architectural patterns. Thus, the patterns concerning the behavioural aspect are not formalized in our study. Representatives of behavioural patterns can be those dealing with invocation mechanism, runtime events, etc. None of the 14 formalized patterns contain a constraint with architecture scope.

The reader can find a complete list of formalized patterns in two paradigms SOA and CBA in Appendix B and Appendix C.

Table 6.3: Categories of architectural patterns from [2]

Architectural view	Pattern	Structural pattern	Pattern with architecture scope constraints
Layered view	2	2	0
Data flow view	2	1	0
Data-centered view	3	3	0
Adaptation view	3	2	0
Language extension view	3	0	0
User interaction view	3	3	0
Component interaction view	5	2	0
Distribution view	3	1	0
Total	24	14	0

6.2.2 StAD documentation

The following discusses the effectiveness and the completeness of our support for StAD documentation. We first introduce the materials used in our study and then go into details of experimental results.

6.2.2.1 Experimental materials

We empirically evaluated our approach with 8 architectural models. These models vary in terms of size and domain. They are gathered from different sources in the literature. These models as well as the applied patterns can be found in Appendix D. We choose Acme as the ADL to depict these models in this experiment but as we stated in Subsection 5.2, it is feasible to change to another ADL since the pattern meta-model is language independent. In Acme, an architecture is described using elements such as *component, connector, port, role, representation, attachment, etc.* among which components and connectors play the most crucial roles. Thus, the size of model is expressed according to three types of measurements: first, by the number of all elements, second, the number of components and third, the number of connectors. Figure 6.6 shows the sizes of the models in terms of model elements, components and

connectors. They differ from small models (49 elements, 7 components and 6 connectors) to big models (287 elements, 34 components and 36 connectors). The models cover different domains, from source code management systems, digital publishing systems to software product line middlewares, etc.

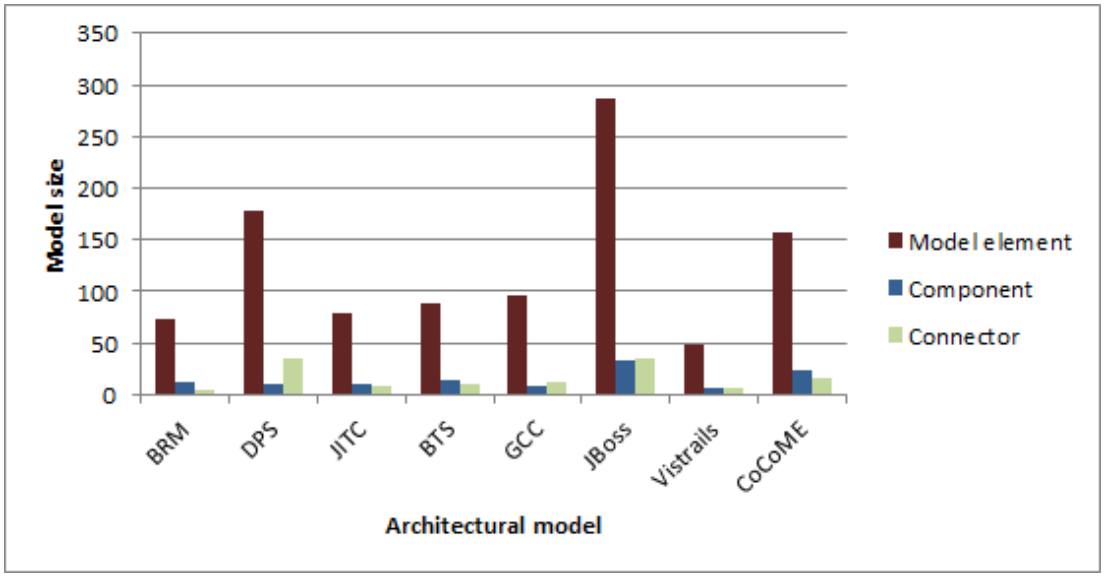

Figure 6.6: Size of 8 Acme architectural models in terms of model elements, components and connectors

One important criterion in choosing these models is that they must focus on the application of architectural patterns in their design. All of the chosen models are indeed designed using architectural patterns from different paradigms such as Enterprise Integration, SOA, etc. On average, there are two patterns applied per model.

6.2.2.2 Modeling effort and StAD violation detection

We present in this section a quantitative evaluation on the modeling effort of using our approach and how this effort is paid off through the detection of StAD violation. As we can recall from section 4.3, the advantage of using mappings is twofold: i) They serve as the bridge between the architectural model and the pattern and thanks to this, the pattern language is independent from any ADL; ii) They are a means to stock the decision of applying a pattern. The question raised is how much effort do we afford to create these mappings for this aim. We count the number of all mappings for each pattern applied in an architectural model and compare it to the number of model elements to determine whether the mappings would not overwhelm the architects. Figure 6.7 shows the size of mappings comparing to size of architectural models in terms of components and connectors. We found that the number of mappings is in average 9.12 (between 3 and 22) and moreover, the average #mappings/#elements is 26.11%, which is

a reasonable number. In fact, seeing that mappings take part in the documentation of StADs, the question of adding mappings or not can be considered as the trade-off of documenting a StAD. This trade-off has been also discussed in [50, 55].

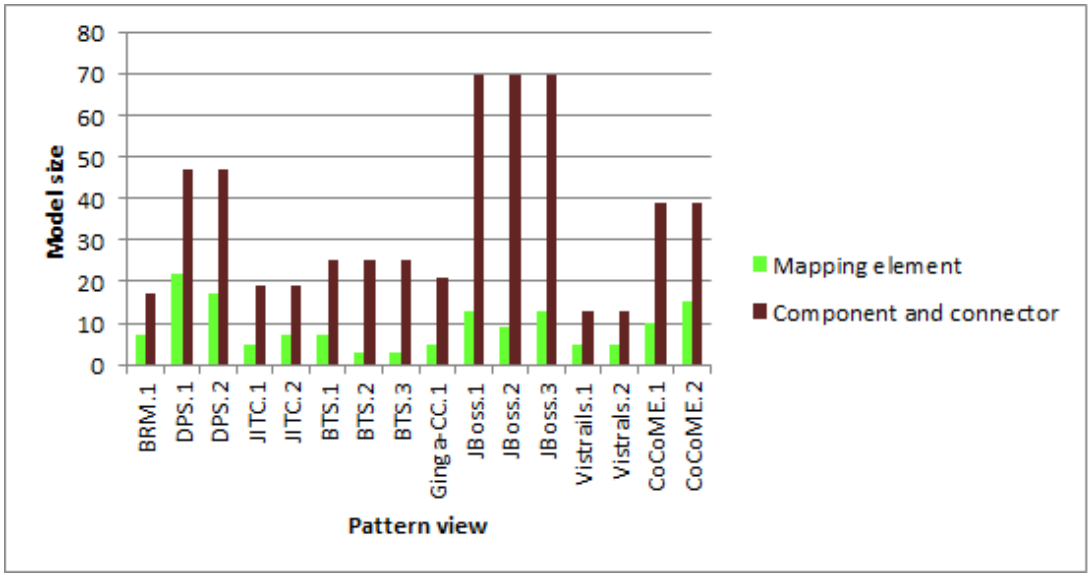

Figure 6.7: Size of mappings comparing to size of architectural models in terms of components and connectors

The first benefit gained from our approach is the obtainment of simplified pattern views. Pattern views serve as a filter of pattern-concerned elements from the architectural model. We empirically evaluated whether pattern views help reduce a great number of non-related elements and thus, improve the understanding of the created StAD. Figure 6.8 shows the comparison between the size of pattern view and the size of the architectural model where it is drawn from. We can observe that most of the pattern views filter out less than 30% of model elements. Two exceptions are the first pattern view of BRM (49 pattern view elements over 73 model elements, equivalent to 67%) and the second view of DPS (91 pattern view elements over 179 model elements, equivalent to 51%). The reasons for this is that the BRM architecture is constructed using *Layers pattern* as the basic principle. Thus, most of the elements participate in the *Layers pattern view*. Similarly, most of the components in the DPS architecture play the role of *Data accessors* in the *Repository pattern*. If we consider all pattern views, pattern view elements are about 25% of the total number of elements in average.

The second benefit of our approach is a complete mechanism of StAD violation detections. Our approach emphasizes the combination of mapping models and pattern models in documenting a StAD. Indeed, mapping models and pattern models together maintain the existence of the StAD and its structural consistency. The absence of one of these two artefacts will lead to an incomplete StAD and thus, undetected violations. To confirm this remark, we make the architectural models evolved and see if we can detect the violations in two cases: i)without

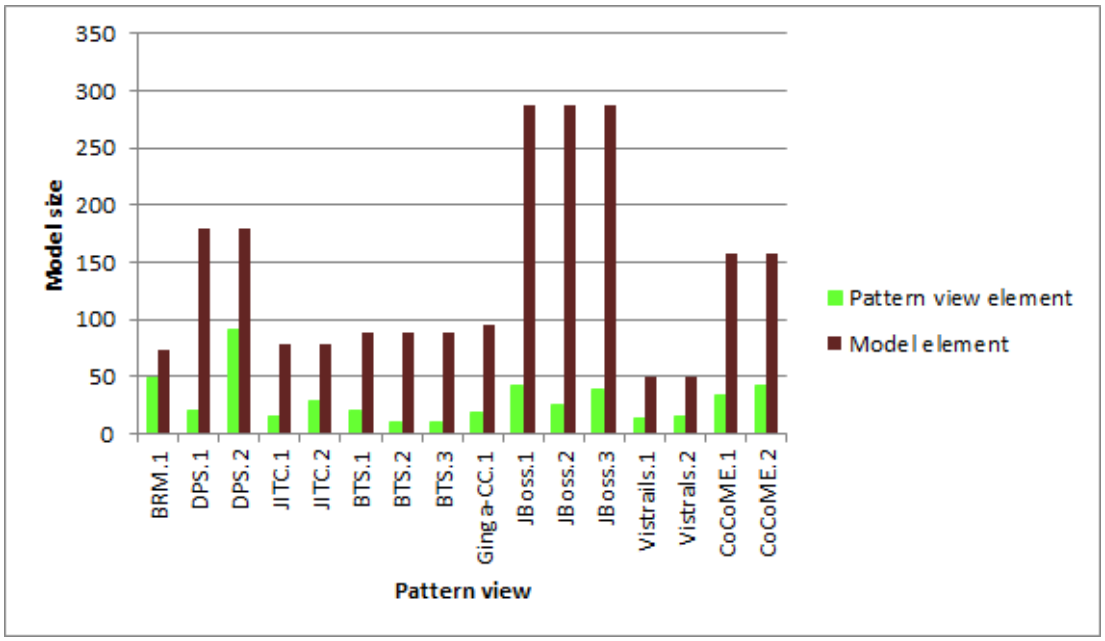

Figure 6.8: Size of pattern view comparing to size of architectural models

mapping models and ii)without pattern models.

An evolution of an architectural model can fall into two cases: deletion and addition of elements (modification is the combination of these two operations). We do not have architects' participation to set up real-life scenarios which involve only meaningful evolutions. Instead, we randomly seed deletion and addition of architectural elements.

Table 6.4 shows the result in the case we delete architectural elements. Theoretically, if N = Nb of components + Nb of connectors, then the total number of combinations of possible deletions is

$$\sum_{k=1}^{N} \binom{N}{k} = 2^N - 1 \tag{6.1}$$

This is in fact the sum of every combination of components and connectors. If we take the case of the biggest model (34 components and 36 connectors), this sum is up to around 10^{21} possible deletions. This will create an exponential explosion. Among these possibles cases, we only need to deal with those that lead to a model conforming to the concerned meta-model. One chosen criterion is to exclude those when the components are deleted but their associated connectors remain. It of course makes no sense of a deletion to leave a dangled connector in the model. Concretely, we choose a subset of meaningful deletions which contains the union of the combinations of connector-related deletions and the combinations of component-related deletions including surrounding connectors. Taking this condition into consideration, the number of possible deletions is reduced significantly and is reported in the column 2 (Nb of combinations of meaningful deletions) of Table 6.4. Particularly, the DPS model has up to 68719478782

Table 6.4: Deletion of architectural elements

Model	Nb of combinations of meaningful deletions	Nb of pattern-related deletions	Nb of detected violations by mapping	Nb of detected violations by pattern	Nb of detected violations by our approach
BRM	157	22	22	16	22
DPS	68719478782	776	776	518	776
JITC	258	32	32	22	32
BTS	3076	30	30	21	30
GCC	4606	10	10	7	10
JBoss	271336	426	426	364	426
Vistrails	190	20	20	14	20
CoCoME	488	460	460	327	460

possible combinations of deletion because its architecture resembles to that of a strongly connected graph where each component has connectors to many other components. Among these deletions, those concerning the application of pattern continue to be filtered out (column Nb of pattern-related deletions). All of these deletions violate the StAD about using pattern (detected by *mapping models or pattern models*). It is clear that 100% violated pattern-related deletions can be detected by mapping models (column Nb of detected violations by mapping) because the latter binds to every element concerning the former. What is noticeable is that there are a certain number of pattern-related deletions that can not be detected by pattern models. As we can observe from the column 5 (Nb of detected violations by pattern), the number of detected violations by pattern models is lower than the total pattern-related violations. In average, 72% of pattern-related violated deletions can be detected by pattern models and the rest 28% can not be detected. The undetected cases of violation by pattern are shown in Appendix E. One example of these could be the case when the first or the last Filters in the Pipes And Filters are deleted. In this case, the remaining Filters and Pipes would make a perfect Pipes And Filters pattern without knowing that some Filters and Pipes have been deleted. In other words, the decision about using the Pipes and Filters pattern has been affected while the pattern model itself cannot recognize it. Another example is the case when the entire pattern (whatever pattern) is deleted. The pattern model is useless since its instances disappeared, leaving the task of keeping track of the decision of using pattern to mapping model. Our approach (last column) detects all violations that were detected by mapping models.

Table 6.5 shows the result in the case we add elements to the architectural models. Since we focus our evaluation on architectural patterns, where the most significant modifications happen

Table 6.5: Addition of architectural elements

Model	Nb of total additions	Nb of pattern-related additions	Nb of detected violations by mapping	Nb of detected violations by pattern	Nb of detected violations by our approach
BRM	30	6	0	3	3
DPS	55	39	0	0	0
JITC	30	9	0	3	3
BTS	57	8	0	5	5
GCC	36	3	0	3	3
JBoss	145	52	0	30	30
Vistrails	21	6	0	3	3
CoCoME	44	34	0	24	24

at a coarse granularity level (component and connector), we consider only additions of component and connector. Among them, we continue to limit the additions to those of connectors between existing components (the combinations of addition are not taken into consideration). The reason for this limitation is that unlike the case of deletions where the number of simulated deletions are finite (seeing that the number of existing elements is fixed), the number of additions is infinite (seeing that we can arbitrarily add elements to the architecture). Besides, the change of an element's definition (name, type, etc.) is considered as the deletion of the element and the addition of a new element (old element with its new definitions). Thus, we took the most basic cases of deletion [1]. The number of the possible additions is reported in column 2 (Nb of total additions). Only a part of these additions relates to patterns. These pattern-related additions are shown in column 3 (Nb of pattern-related additions). Column 4 (Nb of detected violations by mapping) shows that none of the violated additions can be detected by mapping. This is true because the integrity of mapping models is always maintained despite of any addition and thus no violation can be detected. Otherwise, the additions that affect the structural consistency of the applied pattern will be detected by pattern model. As shown in column 5 (Nb of detected violations by pattern), pattern models can detect a part of violated pattern-related additions. In average, 53% of pattern-related additions are violated and detected by pattern models. The violated cases of addition are shown in Appendix E. One example of them could be when we add a Pipe between two distant Filters to create a cycle which is not permitted in Pipes and Filters pattern. Our approach (last column) detects all violations that were detected by pattern models.

[1] This is also discussed as a threat to validity in Section 6.2.2.3

The two above experiments show that depending on the applied patterns, there are violations that can only be detected by the mappings but not by the pattern model and vice-versa. This is also the point that makes our approach stand out from the existing works which focus either only on the existence of AD's element or the structural consistency of AD's element. We combine both mapping, which assures the existence of AD's element, and pattern model, which assure the structural consistency of AD's element, to verify StAD and thus, all violations are detected.

For the purpose of providing enough information to replicate the evaluation, in the following we go to some details with examples about the evaluating process as well as the measurements. Considering the case of the first evaluation (e.g. the deletion seeding evaluation), we first proceed with the measurement of the number of combination of meaningful deletions of a model. A meaningful deletion is one that involves either a connector or a component and does not leave a dangled connector. Since a connector is always connected to two components, the deletion of a single component will always leave at least one connector dangled. Thus, a meaningful deletion of a component must affect all of its connectors. Figure 6.9 shows an example of a simple architectural model (on top of the figure) with three components: *Comp 1*, *Comp 2* and *Comp 3* and two connectors: *Con A*, *Con B*. On the bottom right of the figure is a meaningful deletion, in which *Comp 2* and its associated connector *Con A* are removed respectively. At the bottom, in the middle is the case when *Comp 1* and its surrounding connectors *Con A* and *Con B* are deleted. Finally, the bottom left of the figure is the case when *Comp 3* and *Con B* are deleted. From these 3 cases of component deletions we can create 7 combinations of component deletions (e.g. three 1-combinations, three 2-combinations and one 3-combination). Plus, from two cases of connector deletions, we can create 3 combinations of connector deletions (e.g. two 2-combinations and one 1-combination).

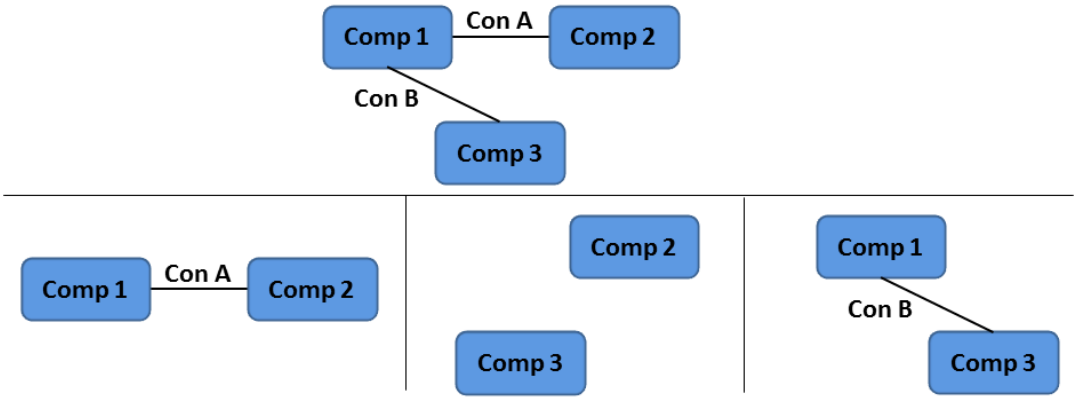

Figure 6.9: Example of meaningful deletions

Thus totally we can create 10 different combinations of deletions (e.g. 3 combinations of connector deletions and 7 combinations of component deletions). On the contrary, if we take

all possible deletions (of components and connectors) into consideration, there are up to 31 combinations of deletions (e.g. from 1-combinations to 6-combinations). This is to see how the filter reduces the number of possible deletion combinations.

Next, two scenarios are set up: the first one only involves mappings in violation detection and the second one only involves pattern model. First, let us assume that our example is a *Client-Server* pattern where *Comp 1* is the Server and *Comp 2, Comp 3* are the Clients. In the first scenario, every element in the model is mapped to the AD. Thus, every deletion of mapped element is considered a violation. For instance, in our example, the deletion of *Comp 2* and *Con A* will be counted as a violation since both of them have been mapped. Thus, three of possible deletion combinations will be detected as violation. In the second scenario, pattern-related elements are directly assigned with roles and the conformance is checked against a pattern meta-model. For instance, the deletion of *Comp 1* will trigger a violation since the pattern constraint is not assured. However the deletion of *Comp 2* and *Con A* will not be detected as a violation since the rest of the model (e.g. *Comp 1*, *Con B* and *Comp 3*) will make a perfect Client-Server pattern. Among the three combinations of deletion, only one is detected (the case when *Comp 2, Comp 3, Con A, Con B* are all deleted and leave no Clients) with the pattern meta-model. Our approach combines these two ways of detecting deletion violation and thus, can detect three cases of violation.

Algorithm 6 The evaluation algorithm

Require: Model M
1: Set of meaningful connector deletions $SetConDels \leftarrow \{\}$ // An empty set of deletions of connectors
2: Set of meaningful component deletions $SetCompDels \leftarrow \{\}$ // An empty set of deletions of components
3: **for all** Element $e \in M.elements$ **do**
4: **if** e is *Connector* **then**
5: $SetConDels \leftarrow e$ // Add the Connector in question to the set of meaningful connector deletions
6: **end if**
7: **if** e is *Component* **then**
8: Set of related elements $SetEles \leftarrow getRelatedElements(e)$ // Derive a set of the Component and its surrounding Connectors
9: $SetCompDels \leftarrow SetEles$ // Add the the set of Component-related elements in question to the set of meaningful deletions
10: **end if**
11: **end for**
12: Set of meaningful deletion combinations $SetDelCombs \leftarrow getCombinations(SetConDels) \cup getCombinations(SetCompDels)$ // Derive the set of combinations of meaningful deletions of connectors and components
13: **for all** Set $s \in SetDelCombs$ **do** // From each combination, perform the deletions and verify the model's conformity
14: Perform the deletions based on s
15: Verify the conformity of the model using mappings
16: Verify the conformity of the model using pattern
17: Verify the conformity of the model using mappings and pattern
18: **end for**

Algorithm 6 shows the skeleton of the evaluation process. From an architectural model, we try first to determine the set of meaningful connector deletions. Next, we determine the

set of meaningful component deletions (for each given component taking into consideration surrounding connectors). Then thanks to a combination generation algorithm[2], we obtain a set of combinations of meaningful connector deletions and a set of combinations of meaningful component deletions. Note that we do not generate all combinations of these two sets because it will create an exponential explosion. Instead, we make a union of these two sets to get a set of meaningful deletions. For each combination of meaningful deletions we apply to the model and verify its conformance using mappings, pattern and both, respectively. Note that the manipulation of model and its elements, and the measurement are done thanks to EMF and its accompanied technologies. We do not show an example of addition seeding and violation detection here but the set-up and applying scenarios remain the same.

6.2.2.3 Threats to validity

This section discusses the study's various threats to validity.

Internal validity: Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variable. In the case of seeding deletion operations, our independent variable is the effect of detecting violations with/without our approach. Similarly, the same measurement is performed in the case of seeding addition operations. We excluded meaningless deletions to reduce the number of treated deletions. However, in case of meaningful deletions and meaningless deletions mixed together, the approach can take more effect. Another point worth mentioning is that we also chose to separate these two independent variables to highlight the drawbacks of using mapping models or pattern models independently. However, in case of seeding deletions and additions one after another, the effect of one independent variable can affect the other one and vice versa.

External validity: In the study, we simulate the architecture's evolution by seeding modifications (deletions and additions). Except for deletion operations when we can determine all possibilities of deleting elements in a model, the addition operations are unpredictable. In the study we treated only additions of connectors between components because the number of these additions is finite. Moreover, deletions and additions are not the only cases of architecture evolution, there are also other types of evolution at the finer granularity level, e.g. elements can be renamed, a connector can change type, and so on. Thus, this experiment cannot generalize the effect of our approach in the evolution of architecture in general. However, the more cases of evolution, the more likely StAD violations are detected by mapping models and pattern models, and thus the more likely the approach has effect.

6.3 Summary

This chapter presented the setup, the process and the result of two empirical evaluations. Basically, they aim at proving the applicability of the two mainstream approaches of the thesis:

[2]We use the Sets.powerSet method in Guava library

the documentation of StAD and the composition of pattern thanks to first-class merging operators. The former consists of seeding architectural modifications in a set of models to show the advantage of the approach comparing to existing ones. The latter consists of building a catalogue of patterns by leveraging first-class status merging operators and showing their benefits. The first evaluation has shown the need for traceability and reconstructablility in pattern composition (which is supported by our approach thanks to documented merging operators). The second evaluation has shown that combining pattern model and mappings is the complete way to detect StAD violation and to avoid missing StAD violation. However, both evaluations still need further validation about its potential extra cost and the acceptance by architects. COMLAN is only applied on a predefined catalogue of well-known patterns, its applicability in reality is not yet validated. Similarly for ADManager, changes in architectural model is simulated by seeding addition and deletion. Thus, real case scenarios still need to involve to complete the validation.

7 Conclusion

This thesis directly deals with two domains: the verification of StAD consistency and the composition of pattern. To address the issue of additional costs caused by the non-explicitness of the ADs, we proposed a solution where the ADs are not only explicit, but also first-class elements in the architecture definition. We specifically focused on ADs about the application of patterns, especially about the use of patterns as solution of structural ADs (StAD). We have shown that the documentation of StADs about the application of patterns should focus on two aspects: the existence of related elements and the structural consistency of the applied patterns. They are complementary aspects and both of them must be considered in evaluating StADs. We pointed out that existing works focus on either one of these two aspects and thus do not detect all possible StAD violation. Our approach aims at leveraging the combination of mapping models and formalized architectural pattern models. This combination brings two major advantages: i) it increases the level of AD reuse during the design stage, ii) it allows to automate the checking of the existence of ADs that must be maintained after the architecture's evolution. More importantly, we empirically show that this is the more complete way comparing to related work to detect StAD violation. We have implemented our approach through a pattern definition language, a process and a tool to automate the checking of the architecture's consistency, with respect to the concerned ADs, during its evolution.

We used MDA, an important approach in model-based software development, to apply our proposal. Models are often used to describe architecture thanks to their high level of abstraction and technology independence. The latter has helped us to make a clear separation between the concepts specific to patterns and those specific to the architecture. This separation aims at making our pattern description language easily adaptable to various ADLs of different paradigms as shown through our empirical evaluation. Other benefits include the conformance checking of the architecture with respect to certain patterns thanks to model validation and the extraction of views targeting the concerned patterns thanks to model transformation.

With our approach architects can build their own library of patterns representing some of its accumulated best practices. However, if one uses different ADLs with different paradigms, or decides to move to an other paradigm, it will take an effort to redefine all existing patterns to fit the new paradigm. There are often common patterns in different paradigms. For instance, the

pattern Pipes and Filters can be found in different paradigms. In SOA, we can cite the example of the free online service Yahoo pipe [1]. In CBA, the architecture of Vistrails architecture presented in Chapter 3 is a good example. This is a limitation in our approach of pattern description. A solution to this limitation would be to describe, in a generic manner, patterns that do not rely on a particular paradigm. Then, provide a means for their projection in each paradigm to avoid the redefinition of pattern. This is one of the perspectives of the thesis.

The use of patterns, when building architecture, has a twofold interest: the use of proven solutions to recurring problems, but also the support for documenting architectural choices. To address the need of documenting pattern-related ADs, especially the ones involving many patterns, we propose a language (COMLAN) for describing architectural patterns and their compositions. This language has the particularity to make explicit the pattern composition operators and the constituent patterns. Making these elements explicit allows us to trace back constituent patterns and in case of changes this way allows the propagation of changes to the composed pattern. Through an empirical study we have shown the importance of these two features for better managing the evolution of architectures.

The use of MDA by our approach not only facilitates the refinement of patterns through the use of transformation models, but also simplifies the definition of the COMLAN language through the use of meta-modeling. Thus we were able to define a meta-model for COMLAN where concepts directly related to the architecture aspect are clearly separated from those related to the pattern aspect. This separation allows an easy adaption of COMLAN to different ADLs, independently of the underlying paradigm (component, service, etc..).

Our pattern description language covers structural aspects of architectures. Thus, patterns that are based on behavioural aspects of an architecture are not supported in our language. Future planned work is to extend our pattern description language to cover also the behavioural aspects of architectures. We found that most of works about the pattern behavioural aspect are based on a variety of behavioural specification mechanisms supported by UML such as automata, petri-net like graph, partially-ordered sequences of event occurrences. These mechanisms are specified through UML diagrams such as state chart, activity, interaction. Thus, elements on pattern languages are normally adapted from those of UML behavioural diagrams. Similarly to these existing pattern languages, our intention is to integrate to COMLAN behavioural elements adapted from UML diagrams. However, the main purpose of integrating the behavioural definition in COMLAN is to support composition. Therefore, there is still work to be done regarding the composition of behavioural aspects in pattern languages.

[1] https://pipes.yahoo.com/pipes/

Bibliography

[1] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, 1997.

[2] Paris Avgeriou and Uwe Zdun. Architectural patterns revisited ,Äì a pattern language. In *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee*, pages 1–39, 2005.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.

[5] Ian Bayley and Hong Zhu. On the composition of design patterns. In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, pages 27–36. IEEE Computer Society, 2008.

[6] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, S Ielceanu, A Miller, A Karmarkar, A Malhotra, J Marino, et al. Sca service component architecture-assembly model specification. *Open Service Oriented Architecture www. osoa. org/download/attachments/35/SCA_Assembly_Model_V100. pdf*, 2007.

[7] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[8] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, 2011.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[10] Rafael Capilla, Francisco Nava, Sandra Pérez, and Juan C. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes*, 2006.

[11] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond (2nd Edition)*. Addison-Wesley Professional, 2010.

[12] Constanze Deiters and Andreas Rausch. A constructive approach to compositional architecture design. In *Proceedings of the 5th European Conference on Software Architecture*, pages 75–82. Springer-Verlag, 2011.

[13] Jing Dong. Representing the applications and compositions of design patterns in uml. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 1092–1098. ACM, 2003.

[14] M Elaasar and L Briand. An overview of uml consistency management. *Carleton University, Canada, Technical Report SCE-04-18*, 2004.

[15] Robert B. France, Dae kyoo Kim, Sudipto Ghosh, and Eunjee Song. A uml-based pattern specification technique. *IEEE Transactions on Software Engineering*, pages 193–206, 2004.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP' 93 - Object-Oriented Programming*, Lecture Notes in Computer Science, pages 406–431. Springer Berlin Heidelberg, 1993.

[17] David Garlan, Robert T. Monroe, and David Wile. Foundations of component-based systems. chapter Acme: architectural description of component-based systems, pages 47–67. Cambridge University Press, 2000.

[18] Imed Hammouda and Kai Koskimies. An approach for structural pattern composition. In *Proceedings of the 6th International Conference on Software Composition*, pages 252–265. Springer-Verlag, 2007.

[19] Neil B. Harrison and Paris Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *Proceedings of the First European Conference on Software Architecture*, pages 263–270, 2007.

[20] ISO/IEC/IEEE 42010:2011. *Systems and Software Engineering - Architecture Description*. ISO, Geneva, Switzerland.

[21] A. Jansen, J. van der Ven, P. Avgeriou, and D.K. Hammer. Tool support for architectural decisions. In *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, pages 4–4, 2007.

[22] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120. IEEE Computer Society, 2005.

[23] Dong Jing, Yang Sheng, and Zhang Kang. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering*, pages 433–453, 2007.

[24] Jung Soo Kim and David Garlan. Analyzing architectural styles. *J. Syst. Softw.*, pages 1216–1235, 2010.

[25] Patrick Könemann and Olaf Zimmermann. Linking design decisions to design models in model-based software development. In *Proceedings of the 4th European Conference on Software Architecture*, pages 246–262. Springer-Verlag, 2010.

[26] Sacha Krakowiak. Middleware architecture with patterns and frameworks, 2007.

[27] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building up and reasoning about architectural knowledge. In *Proceedings of the Second international conference on Quality of Software Architectures*, pages 43–58. Springer-Verlag, 2006.

[28] Martin Küster. Architecture-centric modeling of design decisions for validation and traceability. In *European Conference on Software Architecture (ECSA)*, pages 184–191, 2013.

[29] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of uml model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[30] Ioanna Lytra, Huy Tran, and Uwe Zdun. Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations. In *Proceedings of the 7th European Conference on Software Architecture*, ECSA'13. Springer-Verlag, 2013.

[31] Anders Mattsson, Björn Lundell, Brian Lings, and Brian Fitzgerald. Linking model-driven development and software architecture: A case study. *IEEE Trans. Softw. Eng.*, 35(1):83–93, January 2009.

[32] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, pages 2–57, 2002.

[33] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, pages 70–93, 2000.

[34] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, pages 264–278. Springer-Verlag, 2005.

[35] O.M.G. Model-driven architecture. http://wwww.omg.org/mda.

[36] OMG. Object Constraint Language, OCL Version 2.3.1, formal/2012-01-01. Technical report, OMG, 2012.

[37] Flavio Oquendo. Pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, pages 1–14, May 2004.

[38] L. Sabatucci, A. Garcia, N. Cacho, M. Cossentino, and S. Gaglio. Conquering fine-grained blends of design patterns. In *Proceedings of the 10th International Conference on Software Reuse: High Confidence Software Reuse in Large Systems*, pages 294–305. Springer-Verlag, 2008.

[39] Juha Savolainen, Juha Kuusela, Tomi Mannisto, and Aki Nyyssonen. Experiences in making architectural decisions during the development of a new base station platform. In Muhammad Ali Babar and Ian Gorton, editors, *ECSA*, Lecture Notes in Computer Science, pages 425–432. Springer, 2010.

[40] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.

[41] Antony Tang, Yan Jin, and Jun Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, pages 918–934, 2007.

[42] Minh Tu Ton That, S. Sadou, and F. Oquendo. Using architectural patterns to define architectural decisions. In *Joint IEEE/IFIP Working Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pages 196–200, 2012.

[43] Minh Tu Ton That, Salah Sadou, Flávio Oquendo, and Isabelle Borne. Composition-centered architectural pattern description language. In *Proceedings of the 7th European Conference on Software Architecture*. Springer-Verlag, 2013.

[44] Minh Tu Ton That, Salah Sadou, Flávio Oquendo, and Isabelle Borne. Preserving architectural pattern composition information through explicit merging operators. *Future Generation Computer Systems*, 2014. to appear.

[45] Minh Tu Ton That, Salah Sadou, Flávio Oquendo, and Régis Fleurquin. Preserving architectural decisions through architectural patterns. *Automated Software Engineering*, 2014. to appear.

[46] Erl Thomas. *SOA Design Patterns*. Prentice Hall, 2009.

[47] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 121–130, 2005.

[48] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. A family of languages for architecture constraint specification. *J. Syst. Softw.*, pages 815–831, 2010.

[49] 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*, pages 31–40, 2011.

[50] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, pages 19–27, 2005.

[51] Rob Wojcik, Felix Bachmann, Len Bass, Paul C. Clements, Paulo Merson, Robert Nord, and William G. Wood. Attribute-Driven Design (ADD), Version 2.0. Technical report, Software Engineering Institute, November 2006.

[52] Andrzej Zalewski, Szymon Kijas, and Dorota Sokolowska. Capturing architecture evolution with maps of architectural decisions 2.0. In *Proceedings of the 5th European Conference on Software Architecture*, pages 83–96. Springer-Verlag, 2011.

[53] Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Inf. Softw. Technol.*, pages 1003–1034, 2008.

[54] Uwe Zdun, Paris Avgeriou, Carsten Hentrich, and Schahram Dustdar. Architecting as decision making with patterns and primitives. In *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, pages 11–18, 2008.

[55] O. Zimmermann. Architectural decisions as reusable design assets. *Software, IEEE*, pages 64–69, 2011.

[56] Olaf Zimmermann, Thomas Gschwind, Jochen Küster, Frank Leymann, and Nelly Schuster. Reusable architectural decision models for enterprise application development. In *Proceedings of the Quality of Software Architectures 3rd international conference on Software architectures, components, and applications*, pages 15–32. Springer-Verlag, 2007.

[57] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank leymann. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 157–166. IEEE Computer Society, 2008.

A

Architectural pattern composition catalogue

This appendix documents possible cases of architectural pattern composition. For each pattern we show the source from which we study the pattern as well as a brief description about the pattern. For the sake of simplicity, we do not document the context, the problem, the solution, the implementation, known uses and consequences of pattern. Instead, we concentrate on the structure of the pattern which plays an important role in our study about pattern composition. Especially, for each combined pattern, we show the constituent patterns from which the pattern is composed. For the pattern variant that is formed from other pattern variants, we only show the source variants, the variant's example itself can be deduced from the examples of source variants.

A.1 Enabled Cycle Component [26, 2, 9]

The pattern states that not non-adjacent components can share their data through their connectors. Figure A.1 shows an example of the Enabled Cycle Component pattern. Component 1 can be connected to Component 3 even though they are not adjacent (Component 2 in the middle).

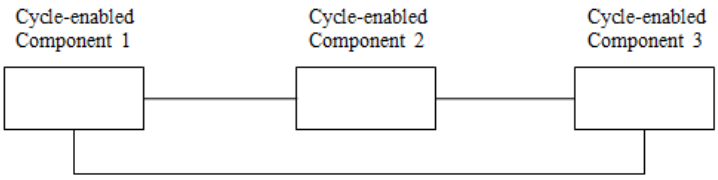

Figure A.1: Cycle enabled component

A.2 Forbidden Cycle Component [2]

The pattern states that only two adjacent components can share data through their connector, but not non-adjacent components. Figure A.2 shows an example of the Forbidden Cycle

Component pattern. Component 1 cannot be connected to Component 3 because they are not adjacent (Component 2 in the middle)

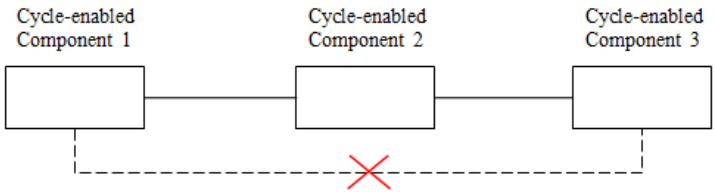

Figure A.2: Forbidden Cycle Component pattern

A.3 Shield [53]

In this pattern, one or more components act as 'shields' for a set of components that form a subsystem. No client should be allowed to access members of the subsystem directly, but access should happen only through these 'shields'. Figure A.3 shows an example of the Shield pattern. Clients can only access to subsystems through shields. Direct access from a client to a subsystem (for instance, from Client 3 to Subsystem 3) is forbidden.

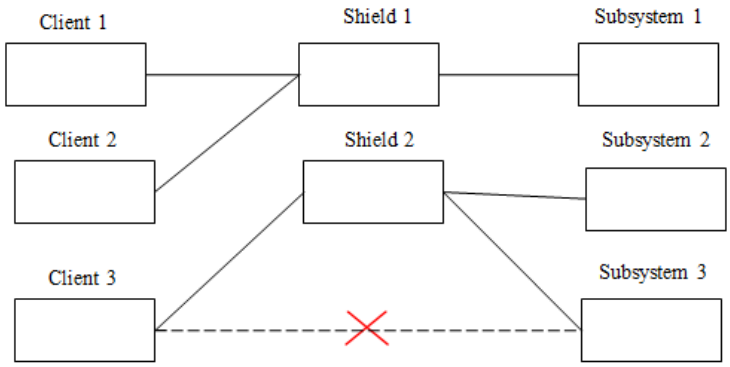

Figure A.3: Shield pattern

A.4 Layers [2, 9]

We consider the most described Layers pattern in the literature as the Basic Layer variant to distinguish from the other variants of Layers pattern.

A.4.1 Basic Layer [2, 9]

In the Basic Layers pattern, the system is structured into Layers so that each layer provides a set of services to the layer above and uses the services of the layer below. Figure A.4 shows an example of the Layers pattern with three layers: Layer 1, Layer 2 and Layer 3

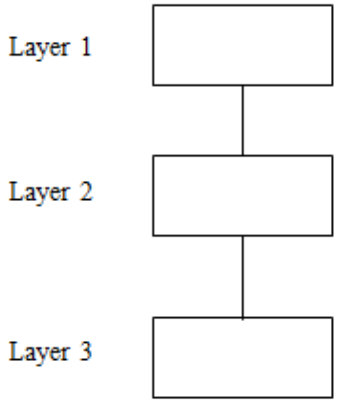

Figure A.4: Basic Layers pattern

A.4.2 By-passed Layers [2]

In this variant of Layers pattern, layers can be by-passed: higher-level layers access lower-level layers without passing through the layer beneath. This variant is the composition of two constituent patterns: the first pattern is Enabled Cycle Component (Section A.1) and the second pattern is Basic Layer (Section A.4.1). The composition is formed by creating the new element Cycle-enabled Layer through the overlapping of the Cycle-enabled Component in the first pattern and the Layer in the second pattern. Figure A.5 shows an example of the By-passed Layers pattern with three layers: Cycle enabled Layer 1, Cycle enabled Layer 2 and Cycle enabled Layer 3. As we can observe, the Cycle enabled Layer 1 can access the Cycle enabled Layer 3 by bypassing the Cycle enabled Layer 2.

A.4.3 Not by-passed Layers [2, 9]

In the pure form of the pattern, layers should not be by-passed: higher-level layers access lower-level layers only through the layer beneath. This variant is the composition of two constituent patterns: the first pattern is Forbidden Cycle Component (Section A.2) and the second pattern is Basic Layer (Section A.4.1). The composition is formed by creating the new element Cycle-forbidden Layer through the overlapping of the Cycle-forbidden Component in the first pattern and the Layer in the second pattern. Figure A.6 shows an example of the Not by-passed

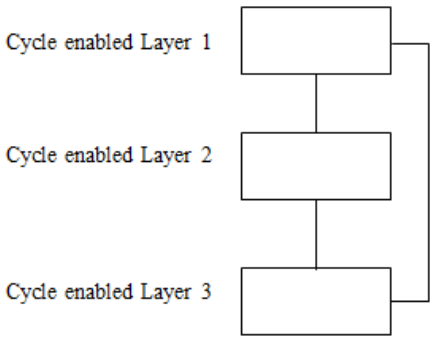

Figure A.5: By-passed Layers

Layers pattern with three layers: Cycle forbidden Layer 1, Cycle forbidden Layer 2 and Cycle forbidden Layer 3. As we can observe, the Cycle forbidden Layer 1 cannot directly access the Cycle forbidden Layer 3 because it cannot bypass the Cycle forbidden Layer 2.

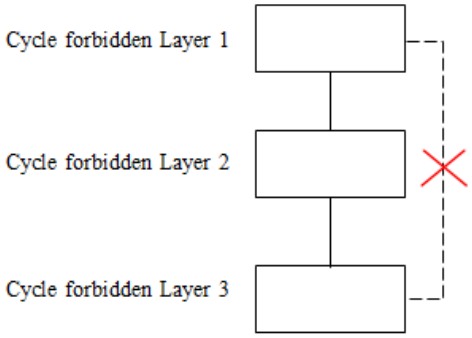

Figure A.6: Not by-passed Layers

A.4.4 Client-Server Layers [2]

In this variant of Layers, two adjacent layers can be considered as a Client-Server pair, the higher layer being the client and the lower layer being the server. This variant is the composition of two constituent patterns: the first pattern is Client-Server (Section A.10) and the second pattern is Basic Layer (Section A.4.1). The composition is formed by creating two new elements: The first new element Client Layer is formed through the overlapping of the Client in the first pattern and the Layer in the second pattern. The second new element Server Layer is formed through the overlapping of the Server in the first pattern and the Layer in the second pattern. Figure A.7 shows an example of the Client-Server Layers pattern with two layers: Client Layer and Server Layer. The Client Layer requests information or services from Server

Layer.

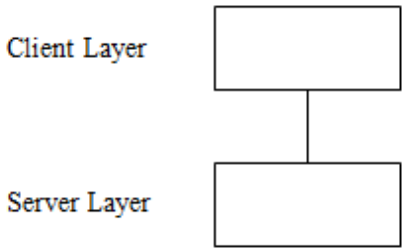

Figure A.7: Client-Server Layers

A.4.5 Filtered Layers [2]

In this variant of Layers, Pipes and Filters can be used for communication between layers, if data flows through layers are needed. This variant is the composition of two constituent patterns: the first pattern is Pipes and Filters (Section refsec1.5) and the second pattern is Basic Layer (Section A.4.1). The composition is formed by creating one new element Filtered Layer which is formed through the overlapping of the Filter in the first pattern and the Layer in the second pattern. Figure A.8 shows an example of the Filtered Layers pattern with two layers: Filtered Layer 1 and Filtered Layer 2. These two layers are connected using a Pipes and Filters pattern. Except for Filtered Layer 1 and Filtered Layer 2, the three other filters are: Filter 1, Filter 2, Filter 3.

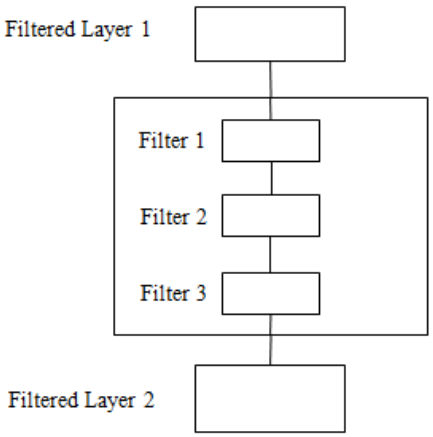

Figure A.8: Filtered Layers pattern

A.5 Pipes and Filters [2, 9]

We consider the most described Pipes and Filters pattern in the literature as the Basic Pipes and Filters variant to distinguish from the other variants of Pipes and Filters pattern.

A.5.1 Basic Pipes and Filters [2][3]

In a Basic Pipes and Filters pattern a complex task is divided into several sequential sub-tasks. Each of these sub-tasks is implemented by a separate, independent component, a Filter, which handles only this task. Figure A.9 shows an example of the Pipes and Filters pattern with three filters: Filter 1, Filter 2 and Filter 3.

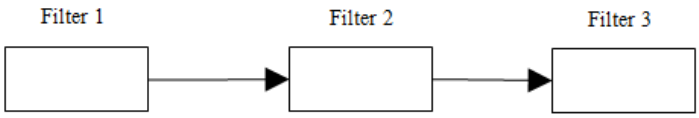

Figure A.9: Basic Pipes and Filters pattern

A.5.2 By-passed Pipes and Filters [2]

In this variant of Pipes and Filters pattern, filters can be by-passed: filters can access non-adjacent filters. This variant is the composition of two constituent patterns: the first pattern is Enabled Cycle Component (Section A.1) and the second pattern is Basic Pipes and Filters (Section A.5.1). The composition is formed by creating the new element Cycle-enabled Filter through the overlapping of the Cycle-enabled Component in the first pattern and the Filter in the second pattern. Figure A.10 shows an example of the Pipes and Filters pattern with three filters: Cycle enabled Filter1, Cycle enabled Filter2 and Cycle enabled Filter 3. As we can observe, the Cycle enabled Filter 1 can access the Cycle enabled Filter 3 by bypassing the Cycle enabled Filter 2.

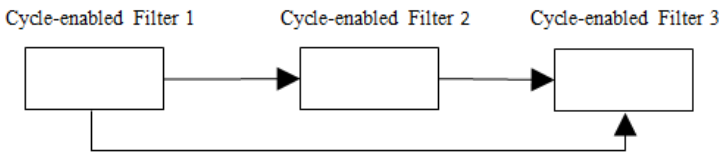

Figure A.10: By-passed Pipes and Filters

A.5.3 Not by-passed Pipes and Filters (or Pipeline) [2, 9]

In this variant of Pipes and Filters pattern, only two adjacent filters can share data through their pipe, but not non-adjacent filters. This variant is the composition of two constituent patterns: the first pattern is Forbidden Cycle Component (Section A.2) and the second pattern is Basic Pipes and Filters (Section A.5.1). The composition is formed by creating the new element Cycle-forbidden Filter through the overlapping of the Cycle-forbidden Component in the first pattern and the Filter in the second pattern. Figure A.11 shows an example of the Not by-passed Pipes and Filters pattern with three filters: Cycle forbidden Filter 1, Cycle forbidden Filter 2 and Cycle forbidden Filter 3. As we can observe, the Cycle forbidden Filter 1 cannot directly access the Cycle forbidden Filter 3 because it cannot bypass the Cycle forbidden Filter 2.

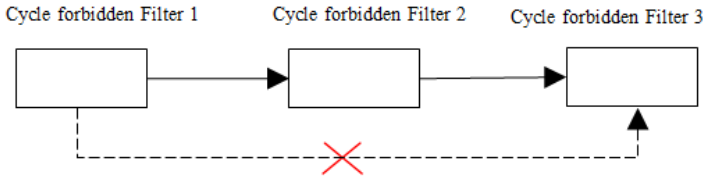

Figure A.11: Not by-passed Pipes and Filters

A.5.4 Internally layer-structured Pipes and Filters [2]

In this variant of Pipes and Filters pattern, the Layers pattern can be used for structuring the internal architecture of Filters. This variant is the composition of two constituent patterns: the first pattern is Layers (Section A.4) and the second pattern is Basic Pipes and Filters (Section A.5.1). The composition is formed by creating the new element Internally layer-structured Filter through the overlapping of the entire first pattern the Filter in the second pattern. Figure A.12 shows an example of the Internally layer-structured Pipes and Filters pattern with three filters: Internally layer-structured Filter 1, Internally layer-structured Filter 2 and Internally layer-structured Filter 3. All these three filters are internally structured by Layer architecture.

A.5.5 Data sharing Pipes and Filters [2]

In this variant of Pipes and Filters pattern, it can be combined with data-centered architectures like Shared Repository to allow for data-sharing between Filters. This variant is the composition of two constituent patterns: the first pattern is Shared Repository (Section refsec1.6.1) and the second pattern is Basic Pipes and Filters (Section A.5.1). In Figure A.13, the composition is formed by creating the new element Internally layer-structured Filter through the overlapping of the entire first pattern the Filter in the second pattern.

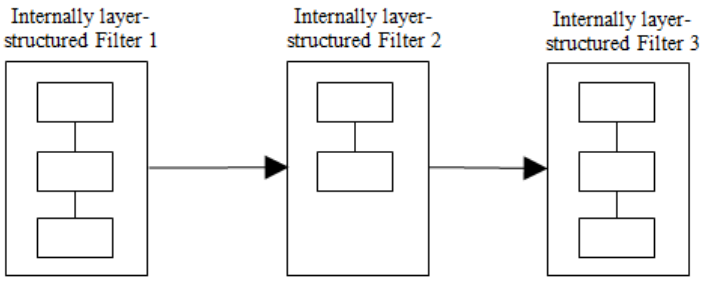

Figure A.12: Internally layer-structured Pipes and Filters pattern

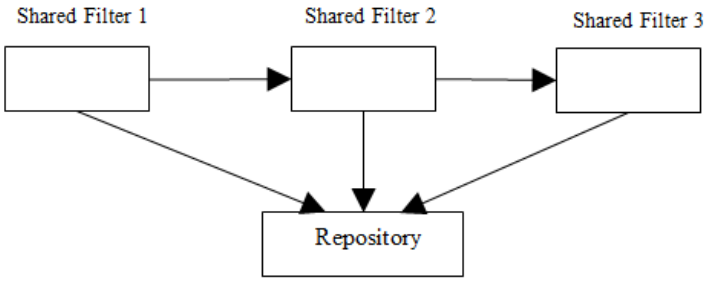

Figure A.13: Data sharing Pipes and Filters pattern

A.6 Shared repository [2, 9, 11]

We consider the most described Shared Repository pattern in the literature as the Basic Shared Repository variant to distinguish from the other variants of Shared Repository pattern.

A.6.1 Basic Shared Repository [2, 9, 11]

In the Shared Repository pattern one component of the system is used as a central data store, accessed by all other independent components. Figure A.14 shows an example of the basic Repository pattern with three Data Accessors: Data Accessor 1, Data Accessor 2 and Data Accessor 3.

A.6.2 Internally Layer structured Shared repository [2]

In this variant of Shared repository pattern, the Layers pattern can be used for structuring the internal architecture of Repository. This variant is the composition of two constituent patterns: the first pattern is Layers (Section A.4) and the second pattern is Basic Repository (Section A.6.1). The composition is formed by creating a new element named Internally layer-structured Repository which is formed through the overlapping of the entire Layers pattern and

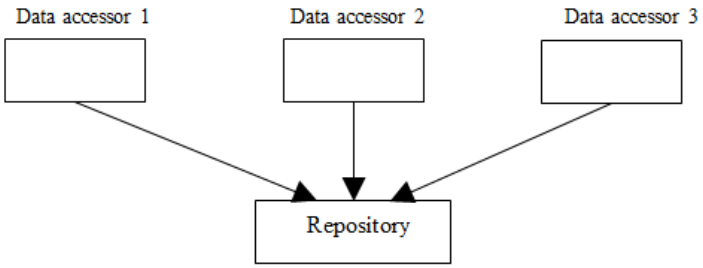

Figure A.14: Basic Repository pattern

the Repository in the Repository pattern. Figure A.15 shows an example of the Internally Layer structured Shared repository pattern with 3 Data accessors: Data accessor 1, Data accessor 2 and Data accessor 3. The Repository is internally structure by a Layers pattern with 3 layers.

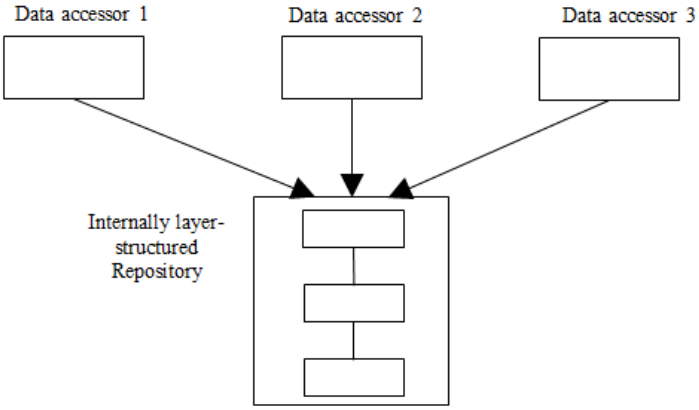

Figure A.15: Internally Layer structured Shared repository pattern

A.7 Microkernel [2]

We consider the most described Microkernel pattern in the literature as the Basic Microkernel variant to distinguish from the other variants of Microkernel pattern.

A.7.1 Basic Microkernel [2]

A Microkernel realizes services that all systems need and a plug-and-play infrastructure for the system-specific services. Internal servers are used to realize version-specific services and they are only accessed through the Microkernel. On the other hand, external servers offer APIs and user interfaces to clients by using the Microkernel. External servers are the only way

for clients to access the Microkernel architecture. Figure A.16 shows an example of the basic Microkernel pattern with one Client, three External Servers, one Microkernel and three Internal Servers.

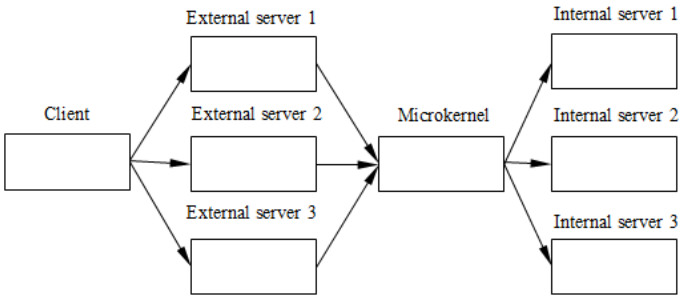

Figure A.16: Basic Microkernel pattern

A.7.2 Broker between Client and External Server [2]

In this variant, Microkernel pattern can be combined with the Broker pattern to hide the communication details between Clients that request services and Servers that implement them. This variant is the composition of two constituent patterns: the first pattern is Broker (Section A.13) and the second pattern is Basic Microkernel (Section A.7.1). The composition is formed by creating two new elements: Client which is formed through the overlapping of the Client in the Broker pattern and the Client in the Microkernel pattern, External Server which is formed through the overlapping of the Server in the Broker pattern and the External Server in the Microkernel pattern. Figure A.17 shows an example of the Broker between Client and External Server pattern.

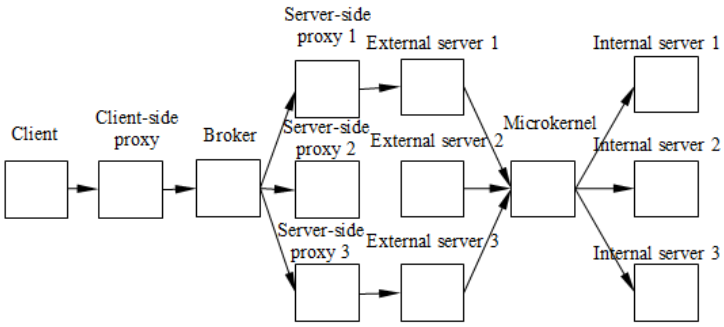

Figure A.17: Broker between Client and External Server pattern

A.8 PAC [2]

The system is decomposed into a tree-like hierarchy of agents. Every agent is designed according to MVC. Figure A.18 shows an example of PAC pattern.

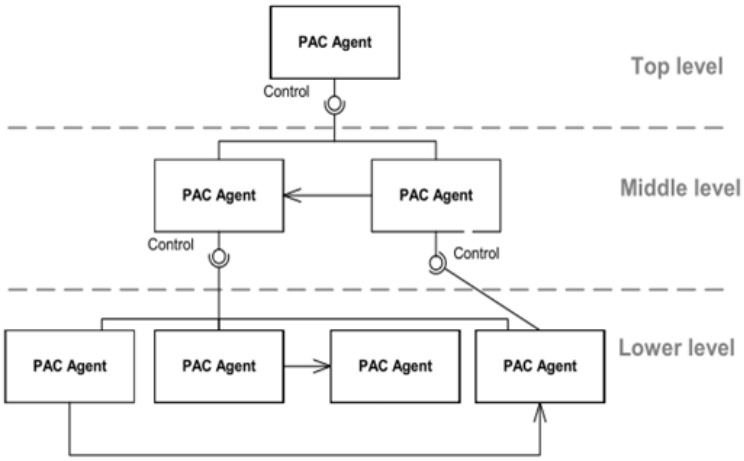

Figure A.18: PAC pattern

A.9 Indirection Layer [2]

An Indirection Layer is a Layer between the accessing client and the the sub-system that needs to be accessed. This variant is the composition of two constituent patterns: the first pattern is Shield (Section A.3) and the second pattern is Indirection Layer itself. The composition is formed by combining i) The Client from the Shield pattern with the Client from the Indirection Layer pattern, ii) The Shield from the Shield pattern with the Indirection Layer from the Indirection Layer pattern, iii) The Sub-system from the Shield pattern with the Sub-system from the Indirection Layer pattern. Figure A.19 shows an example of the Indirection Layer pattern.

A.10 Client-Server [2, 9, 11]

We consider the most described Client-Server pattern in the literature as the Basic Client-Server variant to distinguish from the other variants of Client-Server pattern.

A.10.1 Basic Client-Server [2, 9, 11]

The Client-Server pattern distinguishes two kinds of components: Clients and Servers. The Client requests information or services from a Server. Figure A.20 shows an example of the

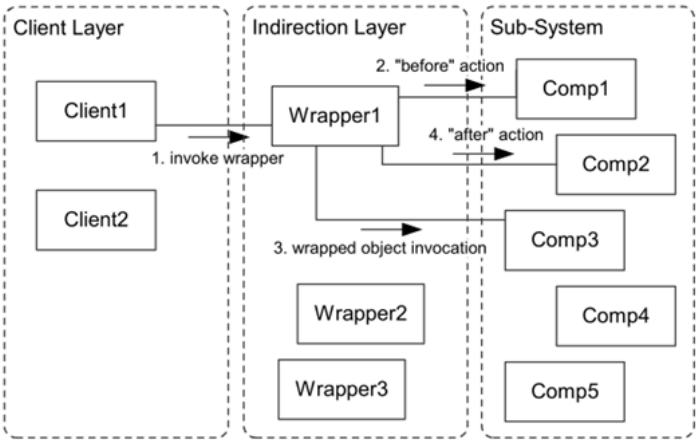

Figure A.19: Indirection layer pattern

Client-Server pattern with three clients: Client 1, Client 2, Client 3.

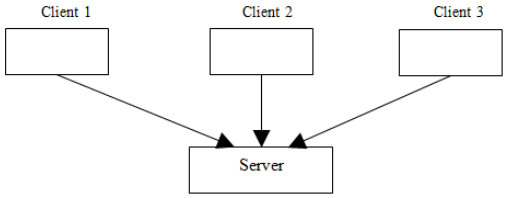

Figure A.20: Client-Server pattern

A.10.2 Client-Server with Broker [2]

Sophisticated, distributed Client-Server architectures usually rely on the Broker pattern to make the complexity of the distributed communication manageable. This variant is the composition of two constituent patterns: the first pattern is Broker (Section A.13) and the second pattern is Basic Client Server (Section A.10.1). The composition is formed by combining i) The Client from the Broker pattern with the Client from the Client-Server pattern, ii) The Server from the Broker pattern with the Server from the Client-Server pattern. Figure A.21 shows an example of the Client-Server with Broker pattern.

A.10.3 Client-Server through Microkernel [2]

In this variant, a Microkernel introduces an indirection that can be useful in certain Client-Server configurations: a client that needs a specific service can request it indirectly through the Microkernel, which establishes the communication to the server that offers this service. In

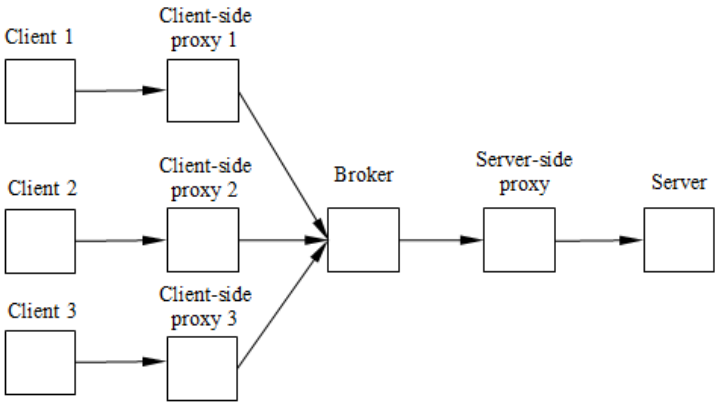

Figure A.21: Client-Server with Broker pattern

this sense all communication between clients and servers is mediated through the Microkernel, for reasons of e.g. security or modifiability. This variant is the composition of two constituent patterns: the first pattern is Microkernel (Section A.7) and the second pattern is Basic Client Server (Section A.10.1). In Figure A.22, the composition is formed by combining i) The Client from the Microkernel pattern with the Client from the Client-Server pattern..

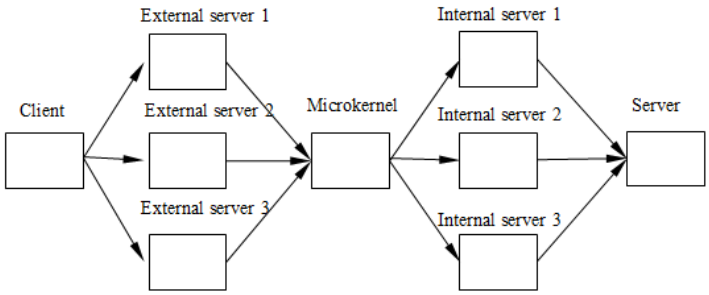

Figure A.22: Client-Server through Microkernel pattern

A.11 MVC [2]

The system is divided into three different parts: a Model that encapsulates some application data and the logic that manipulates that data, independently of the user interfaces; one or multiple Views that display a specific portion of the data to the user; a Controller associated with each View that receives user input and translates it into a request to the Model. Figure 23 is an example of MVC pattern

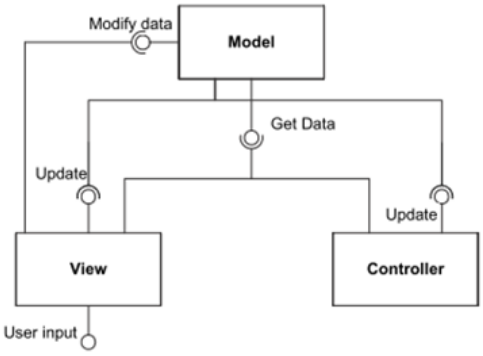

Figure A.23: MVC pattern

A.12 Proxy [9]

This pattern let the client communicate with a representative rather than the component itself. This representative-called a proxy-offers the interface of the component but performs additional pre- and post- processing. This variant is the composition of two constituent patterns: the first pattern is Shield (Section A.3) and the second pattern is Proxy itself. The composition is formed by combining i) The Client from the Shield pattern with the Client from the Basic Proxy pattern, ii) The Shield from the Shield pattern with the Proxy from the Proxy pattern and iii) The Sub-system from the Shield pattern with the Original from the Proxy pattern. Figure A.24 shows an example of the Proxy pattern.

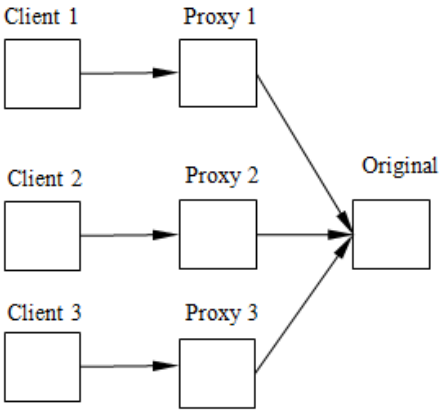

Figure A.24: Proxy pattern

A.13 Broker [2, 9]

In the Broker pattern, a Broker decouples Clients from Servers. Servers register themselves with the Broker, and make their services available to Clients through method interfaces. Clients access the functionality of servers by sending requests via the Broker. The Broker is the composition of 2 Proxy pattern, one from Client to Broker and the other from Broker to Server. Figure A.25 shows an example of Broker pattern. The proxies are applied on the client side and the server side.

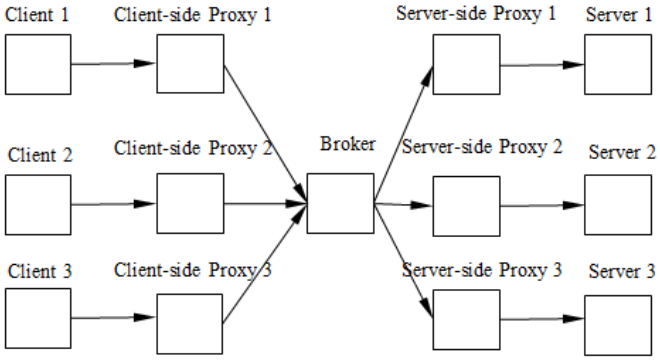

Figure A.25: Broker pattern

A.14 Façade [53, 11]

A façade component is used to abstract a part of the component architecture with negative coupling potential to legacy components. All legacy components must go through the Façade to communicate with the component This variant is the composition of two constituent patterns: the first pattern is Shield (Section A.3) and the second pattern is Basic Façade itself. The composition is formed by combining i) The Client from the Shield pattern with the Legacy Component from the Basic Façade pattern, ii) The Shield from the Shield pattern with the Façade from the Façade pattern. Figure A.26 shows an example of Façade pattern.

A.15 Legacy Wrapper [11]

In this pattern, legacy components are encapsulated with wrappers to insure a seamless communication. This variant is the composition of two constituent patterns: the first pattern is Shield (Section A.3) and the second pattern is Basic Legacy Wrapper itself. As shown in Figure A.27, the composition is formed by combining i) The Client from the Shield pattern with the Consumer from the Basic Legacy Wrapper pattern, ii) The Shield from the Shield

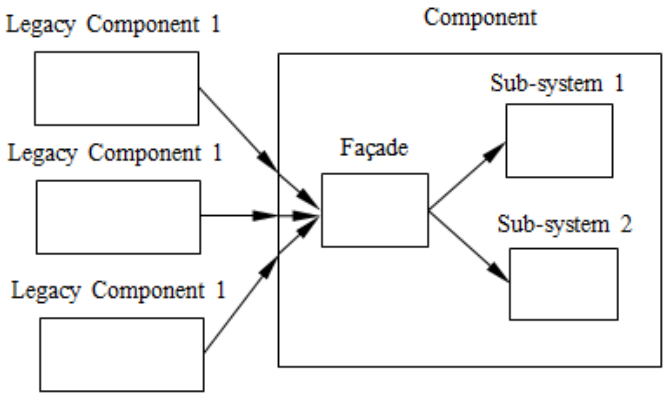

Figure A.26: Façade pattern

pattern with the Legacy Wrapper from the Basic Legacy Wrapper pattern, iii) The Sub-system from the Shield pattern with the Legacy Component from the Basic Legacy Wrapper pattern.

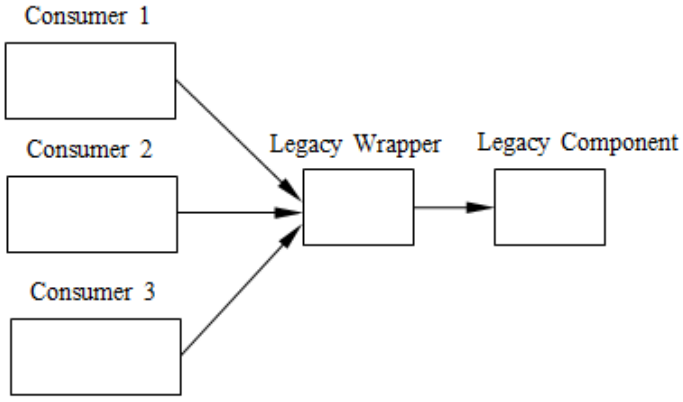

Figure A.27: Legacy Wrapper pattern

A.16 Pipes and Filters + Repository [11]

In this pattern, the sink filter in the Pipes and Filters pattern is connected to the Repository in the Repository pattern to perform data reading/writing operations. This variant is the composition of two constituent patterns: the first pattern is Pipes and Filters (Section A.5) and the second pattern is Repository (Section A.6). The composition is formed by a stringing composition: a new connector (playing the role of both the Pipe in the Pipes and Filters pattern and the Data reading/writing connector in the Repository pattern) is added to connect the first and the second pattern. Figure A.28 shows an example of the Pipes and Filters + Repository pattern.

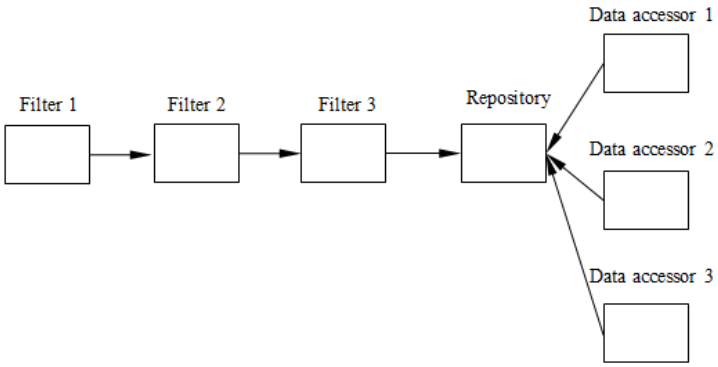

Figure A.28: Pipes and Filters + Repository pattern

B

Formalized SOA pattern

Table B.1 shows the list of formalized SOA patterns along with the number of architectural elements, the involved roles and the number of multiplicity elements.

Table B.1: List of formalized SOA patterns

Pattern category	Patterns	Nb of elements	Roles	Nb of multiplicities
Service inventory	Rules Centralization	2	Service, Rule service	1
	Dual Protocols	1	Protocol	0
	Service Grid	1	Service	0
	Inventory Endpoint	3	Internal inventory service, Inventory endpoint, External consumer	2
	State Repository	2	Service, State repository	1
Service	Service Façade	3	Service, Façade, Consumer	0
	Service Data Replication	2	Service, Replicated database	1
	Partial State Deferral	2	Service, Deferral state repository	0
	Partial Validation	2	Service, Data validator	0

	Decoupled Contract	2	Service, Service contract repository	0
	Legacy Wrapper	2	Legacy Component, Wrapper Component	1
	Exception Shielding	2	Service, Exception Shield	0
	Message Screening	2	Service, Message screener	0
	Trusted Subsystem	1	Service, Trusted Subsystem	0
	Service Perimeter Guard	2	Internal service, Perimeter service	1
	Proxy Capability	2	Service, Proxy	0
	Decomposed Capacity	2	Service, Decomposed proxy	1
	Canonical Protocol	1	Service with uniform protocol	0
	Redundant Implementation	1	Redundant service	1
	Messaging Bridge	2	Message System, Message Bridge	0
	Message Bus	2	Message Bus, Application	1
Service composition	Intermediate Routing	2	Service, Intermediate logic router	0
	Asynchronous Queuing	3	Service, Intermediary buffer, Consumer	0
	Brokered Authentication	3	Service, Broker, Consumer	1

	Data Format Transformation	3	Service, Intermediary data formatter, Legacy component	1
	Service Agent	1	Service Agent	0
	Agnostic Sub-controller	2	Service, Sub-controller	0

C

Formalized CBA pattern

Table C.1 shows the list of formalized CBA patterns along with the number of architectural elements, the involved roles and the number of multiplicity elements.

Table C.1: List of formalized CBA patterns

Pattern category	Patterns	Nb of elements	Roles	Nb of multiplicities
Layered	Layers	2	Layer, Layer connector	1
	Indirection Layer	4	Client layer, Indirection layer, Sub-system, Layer connector	0
Data flow	Pipes and Filters	2	Filter, Pipe	1
Data-centered	Shared Repository	3	Client, Repository, Data accessor	1
	Active Repository	3	Client, Active repository, Data accessor	1
	Blackboard	3	Blackboard, Knowledge source, Data accessor	1

Adaptation	Microkernel	5	Client, External server, Micro kernel, Internal server, Layer connector	2
	Interceptor	4	Client layer, Interceptor, Sub-system, Layer connector	0
User interaction	Model-View-Controller	4	Model, View, Controller, MVC connector	0
	Presentation-Abstraction-Control	2	PAC agent, PAC connector	0
	C2	2	Component, Connector	0
Component interaction	Client-Server	3	Client, Server, Request/Reply connector	1
	Peer to peer	2	Peer, Peer connector	1
Distribution	Broker	4	Client, Server, Broker, Broker connector	0

D

List of architectural models

Table D.1 shows the list of architectural models used in the evaluation along with the applied patterns and their frequency.

Table D.1: List of architectural models

Models	Description	Nb of components	Nb of connectors	Applied patterns	Frequency
BRM	A revenue management system	12	5	Layers	1
DPS	A digital publishing system	11	36	Repository	2
JITC	A source code comprehension aiding system	10	9	Pipes and Filters	1
				Repository	1
BTS	A bond trading system	15	10	Pipes and Filters	1
				Message Bridge	2
GCC	A digital TV system middle-ware	9	12	Pipes and Filters	1
JBoss	An open source J2EE implementation	34	36	Broker	1
				Microkernel	1
				Pipes and Filters	1
Vistrails	A data exploration and visualization open-source system	7	6	Pipes and Filters	1

				Repository	1
CoCoME	A supermarket sales system	23	16	Layers	1
				Message Bus	1

E

Undetected cases of StAD violations

This appendix documents basic reasons for undetected cases of StAD violations which are discovered during the evaluation. As shown in Table E.1, for each pattern encountered during the evaluation, we show the reasons for undetected StAD violation in two cases: using only pattern model and using only mappings.

Table E.1: Undetected cases of StAD violations

Pattern	Violations detected by mappings but not by pattern model	Violations detected by pattern model but not by mappings
Layers	The case when top or bottom layers and connectors are deleted The case when both top and bottom layers and connectors are deleted The case when the entire layers are deleted	The case when a connector between two components that are not adjacent is added
Repository	The case when the DataAccessor and its Read/Write connector are deleted The case when the entire Repository pattern is deleted	
Pipes & Filters	The case when first or last Filters and their Pipes are deleted	The case when a Pipe between two distant Filters is added to create a cycle

	The case when both first and last Filters and their Pipes are deleted The case when the entire Pipes And Filters are deleted	The case when a Pipe between two adjacent Filters that goes in an opposite direction than the other Pipes is added
Message Bridge	The case when the entire Message Bridge pattern is deleted	
Broker	The case when we delete the Client together with its connector to Broker The case when the entire Broker pattern is deleted	The case when a connector between the Remote Object and the Client is added
Microkernel	The case when we delete the External Service together with its connector to Microkernel The case when we delete the Internal Service together with its connector to Microkernel The case when the entire Microkernel is deleted	The case when we add a connector between the External Service and the Internal Service
Message Bus	The case when we delete the Application component and its associated Communicator connector The case when we delete the entire Message Bus	The case when we add a connector between the Application components

Résumé

Les décisions architecturales ont émergé comme un moyen important pour maintenir la qualité de l'architecture pendant sa évolution. L'une des décisions les plus importantes faite par l'architecte sont celles à propos des approches de conception, à savoir des patrons ou des styles architecturales. La structure de ce genre de décision donne la possibilité d'être contrôlée automatiquement. Dans la littérature, il existe quelques travaux sur la vérification automatique de la violation des décisions architecturales. Dans cette thèse, nous montrons que ces travaux ne permettent pas de détecter toutes les violations possibles. Pour les compléter, nous proposons une approche qui i) décrit les patrons architecturaux qui contiennent la définition de la décision architecturale, ii) intègre les décisions architecturales au modèle architectural et iii) automatise la vérification de la conformance de la décision architecturale. Notre approche est implémentée en utilisant EMF et ses technologies accompagnées. Nous avons montré la possibilité de formaliser tous les patrons structuraux. A travers de deux expérimentes, nous avons montré que les décisions architecturales sont bien expliquées et toutes les violations sont détectées.

Les systèmes logiciels composables sont prouvés capable de supporter l'adaptation aux nouvelles exigences grâce à leur flexibilité. Une méthode typique pour composer ces systèmes est de sélectionner et combiner des patrons qui adressent aux exigences de qualité attendues. Plusieurs propositions ont montré l'intérêt de la composition de patron. En revanche, l'un des défauts de ces propositions est la vaporisation de l'information de composition qui conduit au problème de la traçabilité et la reconstructibilité des patrons. Cette thèse a pour but de réserver le statut première classe aux opérateurs de composition pour stocker l'information de composition. L'approche est implémentée dans un outil et une étude empirique a été aussi conduite pour souligner ses intérêts.

Mots-clé: Décision architecturale, Patron, Composition de patron, Ingénierie dirigée par des modèles.

Abstract

Architectural decisions have emerged as a means to maintain the quality of the architecture during its evolution. One of the most important decisions made by architects are those about the design approach such as the use of patterns or styles in the architecture. The structural nature of this type of decisions give them the potential to be controlled systematically. In the literature, there are some works on the automation of architectural decision violation checking. In this thesis we show that these works do not allow to detect all possible architectural decision violations. To solve this problem we propose an approach which: i) describes architectural patterns that hold the architectural decision definition, ii) integrates architectural decisions into an architectural model and, iii) automates the architectural decision conformance checking. The approach is implemented using Eclipse Modeling Framework and its accompanying technologies. Starting from well-known architectural patterns, we show that we can formalize all those related to the structural aspect. Through two experiments, we show that architectural decisions are well explained and all of their violations are detected.

Composable software systems have been proved to support the adaptation to new requirements thanks to their flexibility. A typical method of composable software development is to select and combine a number of patterns that address the expected quality requirements. A lot of work have shown the interest of pattern composition. Nevertheless, one of the shortcomings of these work is the vaporization of composition information which leads to the problem of traceability and reconstructability of patterns. This thesis also proposes to give first-class status to pattern merging operators to facilitate the preservation of composition information. The approach is tool-supported and an empirical study has also been conducted to highlight its interests.

Keywords: Architectural decision, Pattern, Pattern composition, Model Driving Engineering.

n d'ordre : 000000000

Université de Bretagne Sud

Centre d'Enseignement et de Recherche Y. Coppens - rue Yves Mainguy - 56000 VANNES

Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70