# Preserving architectural pattern composition information through explicit merging operators

M.T.T. That, S. Sadou, Flavio Oquendo, I. Borne

# Preserving architectural pattern composition information through explicit merging operators

Minh Tu Thon That, S Sadou, Flavio Oquendo, I Borne

# Preserving architectural pattern composition information through explicit merging operators

M.T.T. That[a], S. Sadou[a], F. Oquendo[a], I. Borne[a]

[a] *Université de Bretagne Sud, IRISA, Vannes, France*

**Abstract**

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. Therefore, pattern composition has become a crucial aspect during software design. One of the shortcomings of existing work about pattern composition is the vaporization of composition information which leads to the problem of traceability and reconstructability of patterns. In this paper we propose 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 effectiveness. By applying the approach on the composition of a set of formalized architectural patterns, including their variants, we have shown that composed patterns have become traceable and reconstructable.

*Keywords:* Architectural pattern, Pattern composition, Model driven engineering

## 1. Introduction

A key issue in the design of any software system is the software architecture. It consists of the fundamental organization of the system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution <14>. Software architecture gives a basis for analysis of software systems before the system has been built and thus, helps manage risk and reduces cost during the software development. A good software architecture produces not only a system that works properly (the functional requirements) but also a system that meets non-functional requirements such as maintainability, exchangeability, reusability, etc. <13>. Patterns address this important objective of software architecture by allowing the construction of specific software architectures with well-defined properties.

*Email addresses:* minh-tu.ton-that@irisa.fr (M.T.T. That), Salah.Sadou@irisa.fr (S. Sadou), Flavio.Oquendo@irisa.fr (F. Oquendo), Isabelle.Borne@irisa.fr (I. Borne)

Indeed, some architecture description languages (ADL) such as Wright and ACME <1; 10> support the construction of architectures by using predefined architectural patterns. Thus, the construction of architectures is simplified (reuse of the whole pattern structure) and equipped with proven solutions for well-known needs. Further, in real world architectures recurring problems are complex and their solutions can be represented by patterns in complex forms that require the combination and reuse of other existing architectural patterns <5>. The combined patterns on one hand, handle the increased complexity of the architecture and on the other hand, capture the properties of participating patterns. For instance, a system might use a pipe and filter pattern to process data but write the result to a shared database. Thus, this system uses a pattern which is the combination of pipe and filter and shared repository patterns. In the literature, current support for pattern composition consists in fact of using merging operators that are not part of the pattern language <11; 3; 23; 7>. Thus, once the architectural solution achieved there is no means to know that it is a result of a composition of patterns. This is caused by what we call the vaporization of composition information. This prevents the traceability as well as the reconstructability of patterns which are essential for software evolution.

For addressing these open issues, we propose to reserve first-class citizenship for pattern merging operators. Throughout this paper, we show that being able to store and manipulate merging operators is crucial in the context of pattern construction. The idea is implemented in an architectural pattern description language, called COMLAN (Composition-Centered Architectural Pattern Description Language). The language provides a proper description of pattern that supports composition operations and a two-step pattern design process that helps to preserve pattern composition information. The idea is applicable in both architectural patterns and design patterns. However, in this work we focus our approach as well as its evaluation on architectural patterns.

The remainder of this paper is organized as follows: Section 2 presents the state-of-the-art for pattern composition and pattern description. Section 3 points out the open issues through examples. Section 4 introduces the general approach to address the identified problems. Section 5 goes into details of the pattern description language. Section 6 describes the pattern refinement step. Section 7 gives implementation information. Section 8 describes the validation of the proposed approach using empirical studies. Finally, section 9 concludes the paper.

## 2. State of the art

As described in the previous section, the research problem we deal with is the vaporization of pattern composition information. This problem directly concerns two research areas: pattern composition and pattern description in ADLs. In the following we will elaborate the state of the art of these domains.

*2.1. Pattern composition*

There are mainly two branches of work on the composition of patterns. The first including <11; 3; 23> 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 the different terminologies used in <11> (conservative composition), in <23> (unification) or in <3> (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 <23> is the composition of the Mediator pattern and the Proxy pattern as shown in Figure 1. 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.

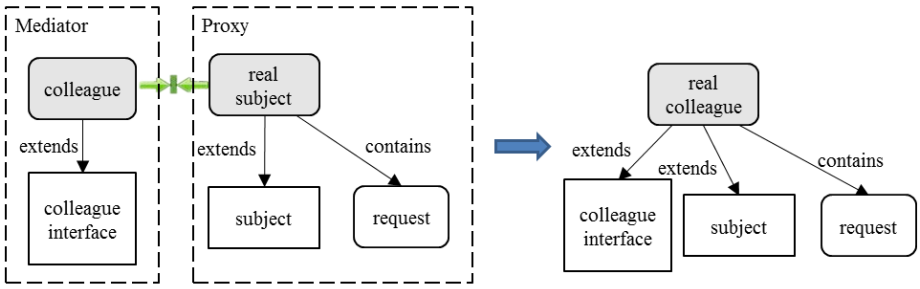


Figure 1: Overlapping composition of Mediator pattern and Proxy pattern

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 <11> or conjunction in <23>. 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, 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.

On the contrary to the approaches above, in <7>, Deiters et al. propose to compose pattern at instance level. An architecture entity can at the same

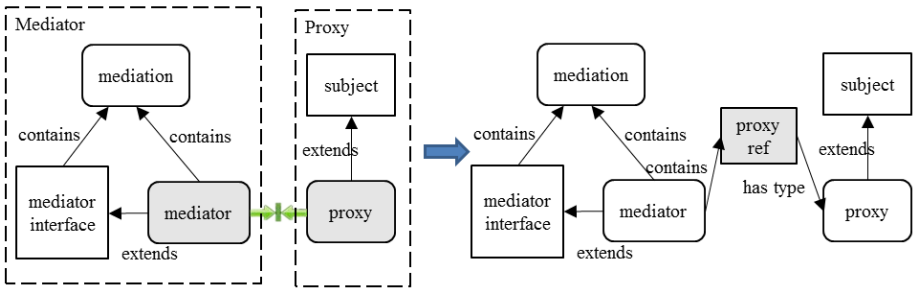


Figure 2: Conjunction composition of Mediator pattern and Proxy pattern

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 <15>, Jing et al. propose a UML profile to attach pattern-related information on merged elements in composed patterns. Figure 3 is an example taken from <15>. 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, <8> 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 <26>, 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 4, which is taken from <26>, 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.

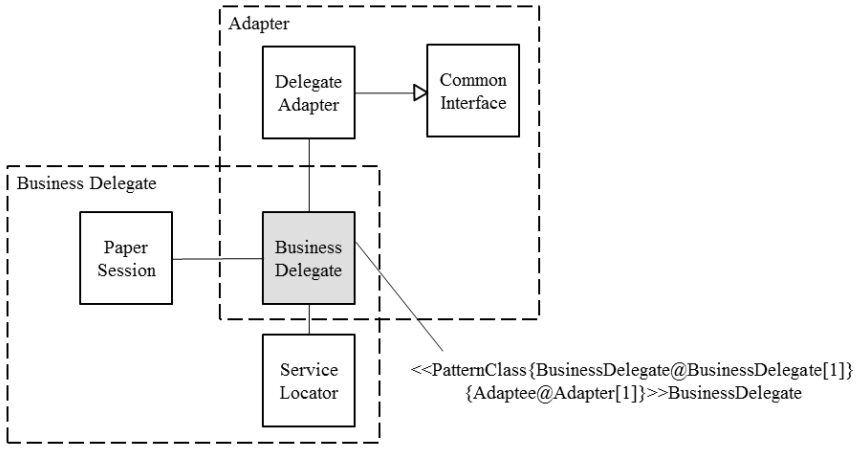


Figure 3: UML profile for attaching pattern composition information

*2.2. 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 <6>. 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 paper 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 <10> 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
...
}
```

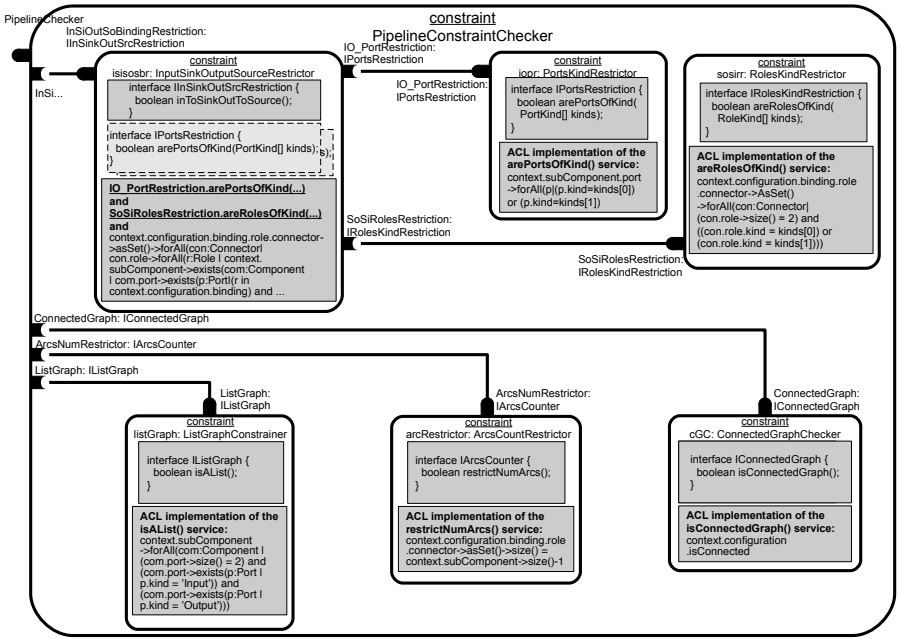


Figure 4: Architectural constraint composition (Figure taken from <26>)

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. In another work, <22> proposes a modelling language based on ontology to formally define architectural pattern. Patterns are designed through a type constraint language and combined using an operator calculus.

As opposed to these domain specific languages, in <17> 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 <27>, 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 5 is the example taken from <27> 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 5. This primitive is also applied in other patterns such as Layer, Façade, etc. which makes it a common composable structure in pattern solutions.

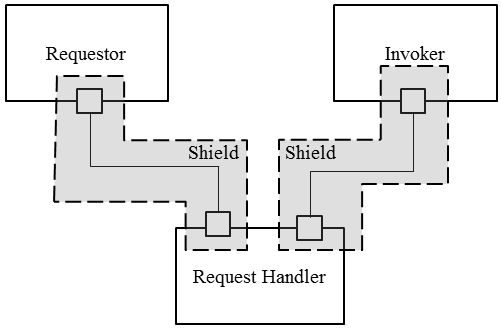


Figure 5: Shield architectural primitive

Architectural patterns and styles are also considered important artefacts that represent architectural knowledge (AK). Among AK-based approaches, works concerning patterns and styles fall into the knowledge reuse category whose main purpose is applying existing knowledge in a particular context for various purposes <29>. Works such as <24; 12; 30> propose to use architectural pattern languages to capture AK. The idea is to leverage reusable design knowledge in patterns to inexpensively document AK in a specific context.

## 3. Research Challenge and Solution

We identify the research challenges via some illustrative examples of architectural pattern composition and introduce the overall approach to solve these challenges.

### *3.1. Problem statement and discussion*

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.

*3.1.1. Blend of patterns*

By observing the documented patterns in <5; 6>, 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.

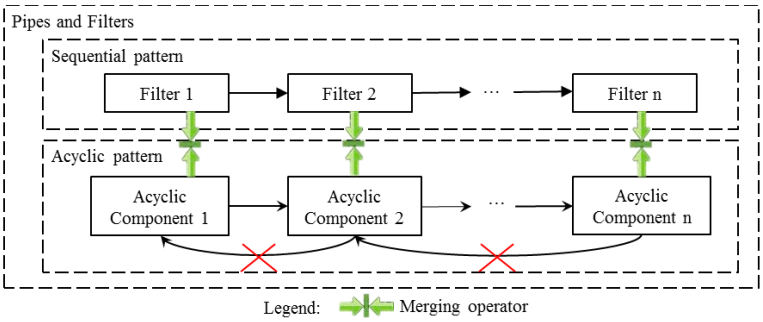


Figure 6: Pipes and Filters

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 6). 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*.

*3.1.2. Connection of patterns*

A lot of documented patterns formed from two different patterns can be found in <6; 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 7.

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.

*3.1.3. 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

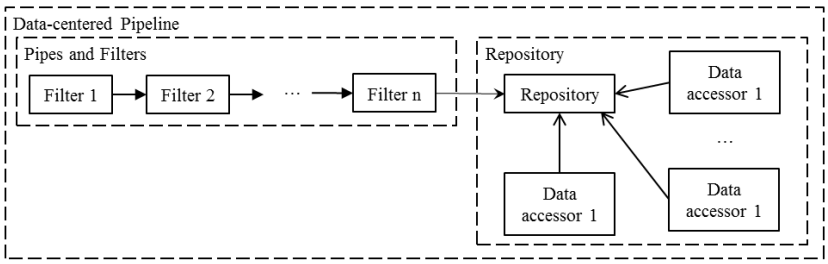


Figure 7: The Data-centered pipeline pattern

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 8. 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.

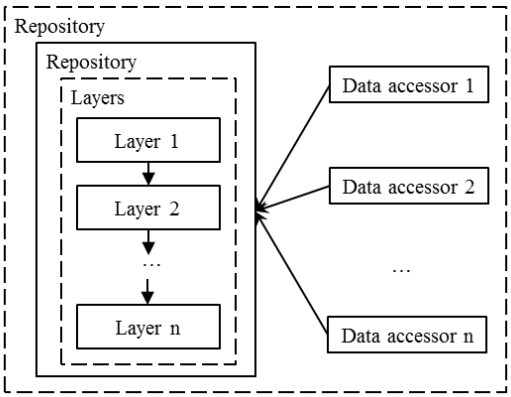


Figure 8: Layers as internal structure of Repository

*3.1.4. Discussion*

As we can observe from the example of subsection 3.1.2, 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 3.1.1. 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 3.1.1, 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 3.1.2 must be also reconstructed. The same requirement exists for the example of subsection 3.1.3. 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, in the literature, the already proposed approaches about pattern composition (see subsection 2.1) 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 <15>, 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.

In the following, we present our approach to address these two problems.

### *3.2. Solution Overview*

We propose the process of constructing patterns including two steps as illustrated in Figure 9. The first step consists in describing a pattern as a composition graph of unit patterns using the COMLAN language see Section 4. Thus, 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 (see Section 4.1), 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 <19>. Each pattern is considered as a model conforming to the COMLAN meta-model (see Sub-section 4.1) 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.

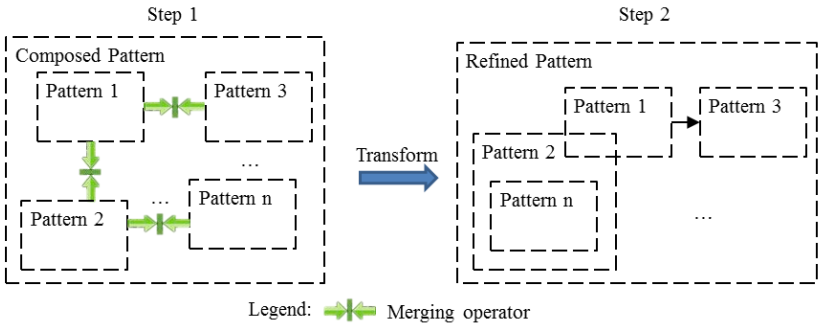


Figure 9: Overall Approach

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 on 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 two following sections, we describe our pattern description language and the transformation process that produces the refined pattern model from a pattern model.

## 4. A pattern description language for hierarchical pattern and composition

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.

### *4.1. The COMLAN meta-model*

In this work, we reuse part of our role-based pattern language <24> which serves for documenting architectural decisions about the application of architectural 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 10, our meta-model is composed of two parts: the structural part and the pattern part. As pointed out in <16; 1> and also described in <6>, 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:

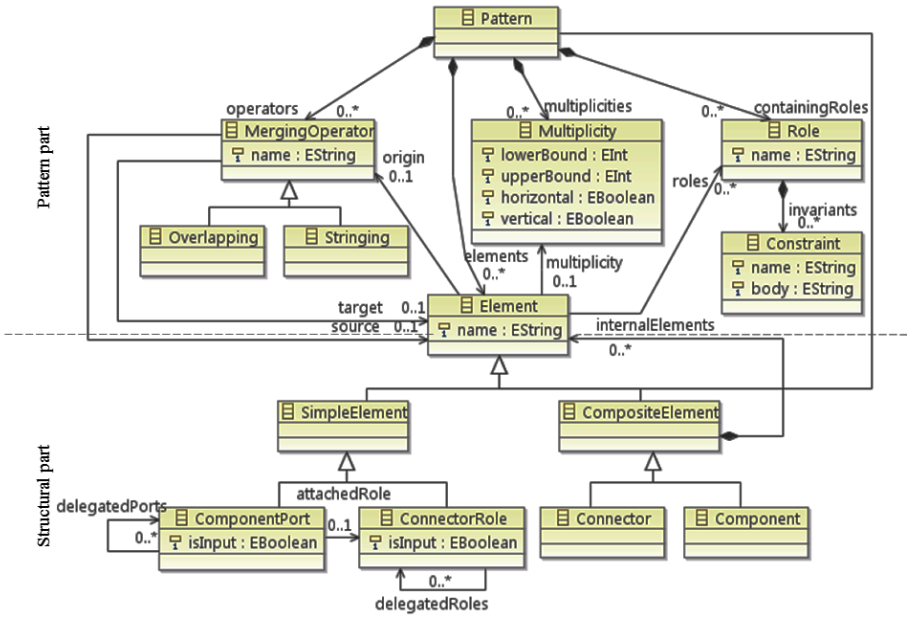


Figure 10: The COMLAN meta-model

- *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, these two types of containment relation are differentiated by a constraint imposed on the meta-model.

- *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 10) 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 11 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 <5>.

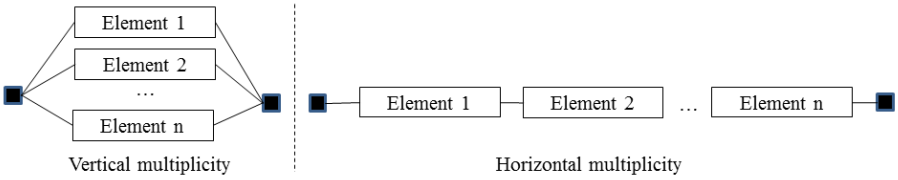


Figure 11: Orientation organization of generic elements

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 <9>. 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 (Object Constraint Language) <20> rules.

Similar to <11; 3; 23>, in our language two types of merging operator are supported: stringing and overlapping as shown in Figure 12. 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.

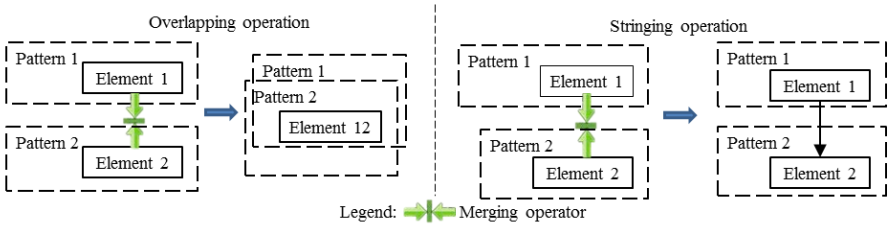


Figure 12: Two types of merging operation

*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.

### *4.2. Pattern definition through an example*

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 <4>. More specifically, this model represents the first step of the pattern definition process. As shown in Figure 13, 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 13, we can observe that the *Pipes and Filters* 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 *Connected-Filter* 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

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

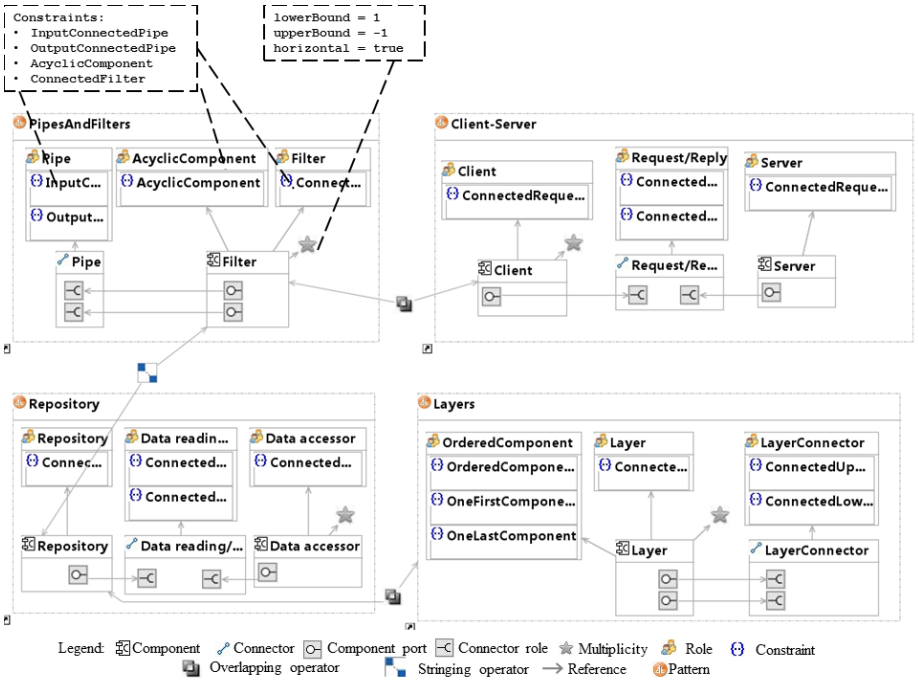


Figure 13: Example of pattern model

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
   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 13), 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.

## 5. 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.

### *5.1. 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 14, 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

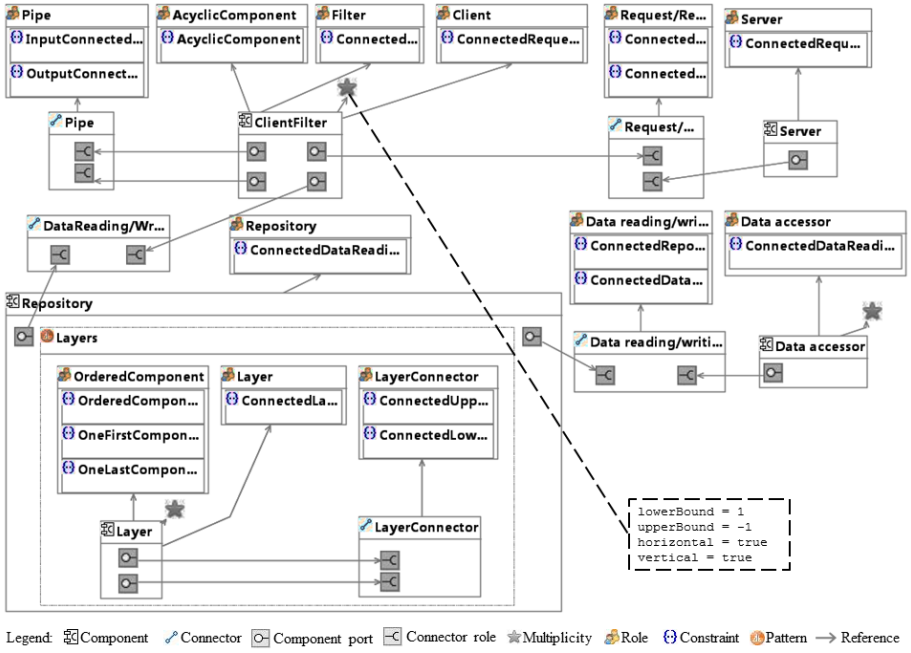


Figure 14: The refined pattern model

### *5.2. 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 14, the overlapping operator described in the previous step is concretized by the component *ClientFilter*. 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.

**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 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 elements multiplicity or the target elements multiplicity is vertical or horizontal then merged elements multiplicity is also vertical or horizontal. In our pattern model (Figure 14), 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 15. A simplified version of the transformation algorithm for overlapping operator is presented in Algorithm 2

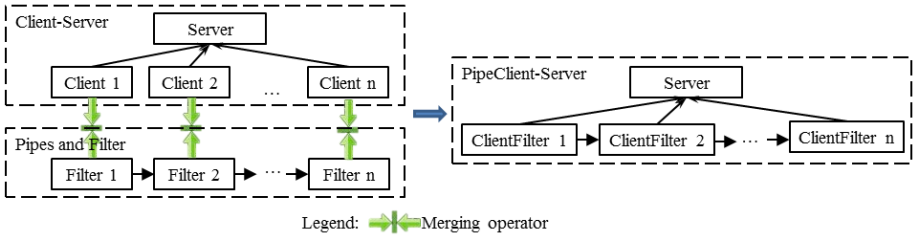


Figure 15: The merged pattern of Client-Server and Pipes and Filters

In the case of a chain of consecutive overlapping operators in which one continues another, we use Algorithm 3. Let's 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

**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: $MergedElem.multiplicity \leftarrow multimerge(SrcElem.multiplicity, TarElem.multiplicity)$

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: $MergedElem \leftarrow overlappingMerge(ElemSet[i], ElemSet[i+1])$
4: $ElemSet[i+1] \leftarrow MergedElem$
5: **end for**

### *5.3. 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 14, 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.

*5.4. Support of traceability and reconstructability*

For every merged element in the composed pattern, the origin reference towards the merging operator helps to preserve information about which element in the constituent pattern participating in the composition process. Figure 16 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. 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.

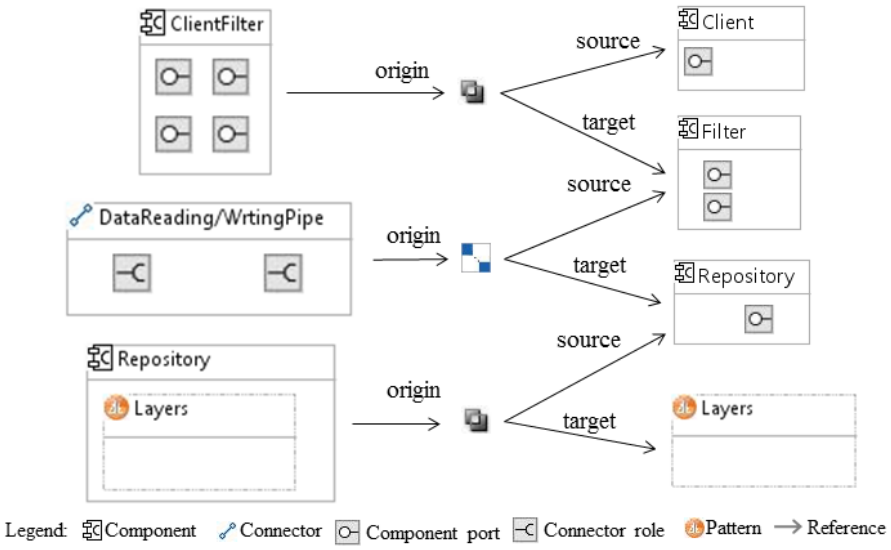


Figure 16: Support of traceability

The support for pattern reconstruction is pretty straightforward. Whenever a constituent pattern is modified, a more complex variant of the Layers pattern is used instead of the pure variant for example, 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.

## 6. Implementation

We developed the *COMLAN* tool, a graphical representation of the COMLAN pattern description language. With the *COMLAN* tool we aim to make

concrete the aforementioned concepts. Thus, this tool provides the following functionalities:

1. Create architectural patterns
2. Compose patterns using merging operators
3. Refine the composed pattern

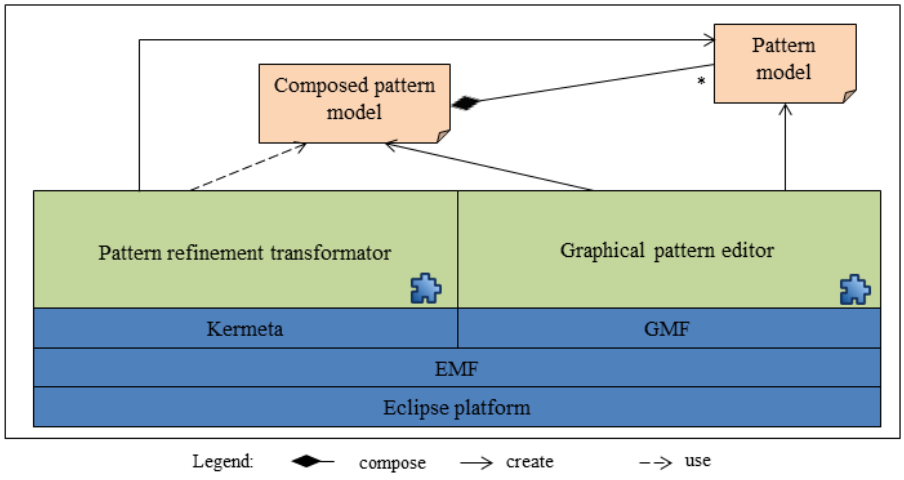


Figure 17: COMLAN tool architecture

*COMLAN* is based on EMF (Eclipse Modelling Framework)[2]. We chose EMF to realize our tool since we leverage MDA, where models are basic building units, to develop our approach. Figure 17 depicts the architecture of the COMLAN tool. The tool consists of two Eclipse plug-ins built on existing Eclipse technologies:

- *Pattern editor plug-in* uses EMF and GMF (Graphical Modeling Framework)[3] 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 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 18 represents several snapshots of COMLAN tool. The bottom-left

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

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.

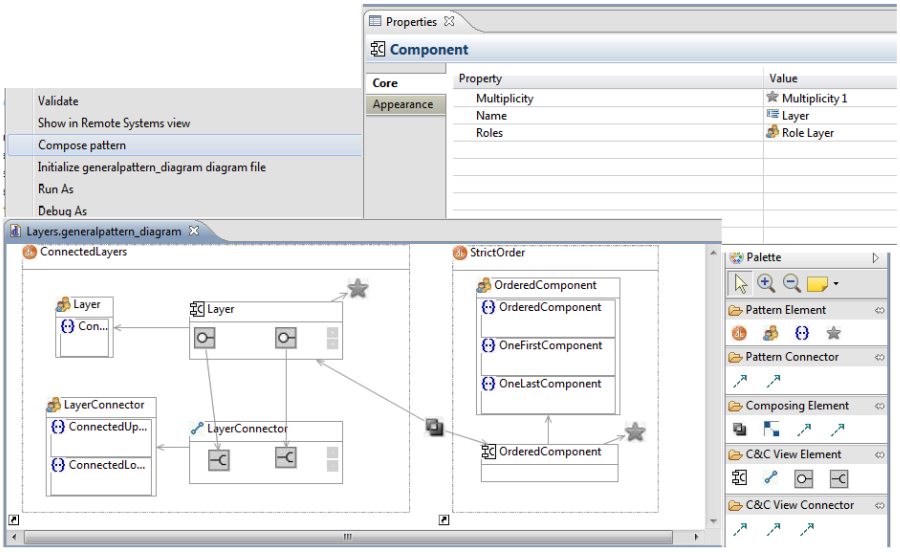


Figure 18: Snapshots of COMLAN tool

- *Pattern refinement plug-in* uses *Kermeta* [4] 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 [5] for a complete tutorial and a video about this tool and the example of Vistrails's architecture <4> pattern modeling.

[4] Kermeta is described in details in <18>

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

## 7. Empirical study

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.

### *7.1. Experimental setup*

The materials used in our experiment are patterns we gather from different sources of architectural patterns in the literature such as <2; 5; 27; 6; 25>. We distinguished two levels of pattern granularity: primitive level and architectural level. As being shown via the Acyclic pattern in the illustrative example (Section 3.1.1), 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 <5; 6> or architectural primitives in <27; 28>. 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 1 shows the catalogue of pattern definitions and their variants. A complete technical report about the catalogue of patterns and variants used in our experiment can also be found at the website of COMLAN[6].

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

[6] `http://www-irisa.univ-ubs.fr/ARCHWARE/software/COMLAN/`

Table 1: Pattern catalogue

| Pattern definitions | Pattern variants |
|---|---|
| P1-Enabled cycle | V1.1-Enabled cycle <2; 5> |
| P2-Forbidden cycle | V2.1-Forbidden cycle <2> |
| P3-Shield | V3.1-Shield <27> |
| P4-Layers | V4.1-Basic Layers <2; 5><br>V4.2-By-passed Layers <2><br>V4.3-Not By-passed Layers <2; 5><br>V4.4-Client-Server Layers <2><br>V4.5-Filtered Layers <2> |
| P5-Pipes & Filters | V5.1-Basic Pipes and Filters <2; 5><br>V5.2-By-passed Pipes and Filters <2><br>V5.3-Pipeline <2; 5><br>V5.4-Layer-structured Pipes and Filters <2><br>V5.5-Data sharing Pipes and Filters <2> |
| P6- Shared Repository | V6.1-Basic Shared Repository <2; 5; 6><br>V6.2-Layer-structured Shared Repository <2> |
| P7-Microkernel | V7.1-Basic Microkernel <2><br>V7.2-Microkernel with Broker <2> |
| P8-PAC | V8.1-PAC <2; 5> |
| P9-Indirection Layer | V9.1-Indirection Layer <2> |
| P10-Client-Server | V10.1-Basic Client-Server <2; 5; 6><br>V10.2-Client-Server with Broker <2><br>V10.3-Client-Server with Microkernel <2> |
| P11-MVC | V11.1-MVC <2; 5> |
| P12-Proxy | V12.1-Proxy <2; 5> |
| P13-Broker | V13.1-Broker <2; 5> |
| P14-Façade | V14.1-Façade <27; 25> |
| P15-Legacy Wrapper | V15.1-Legacy Wrapper <6> |
| P16-Data-centered Pipes and Filters | V16.1-Data-centered Pipes and Filters <6> |

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 structures

**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.

### *7.2. 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 19 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 20 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

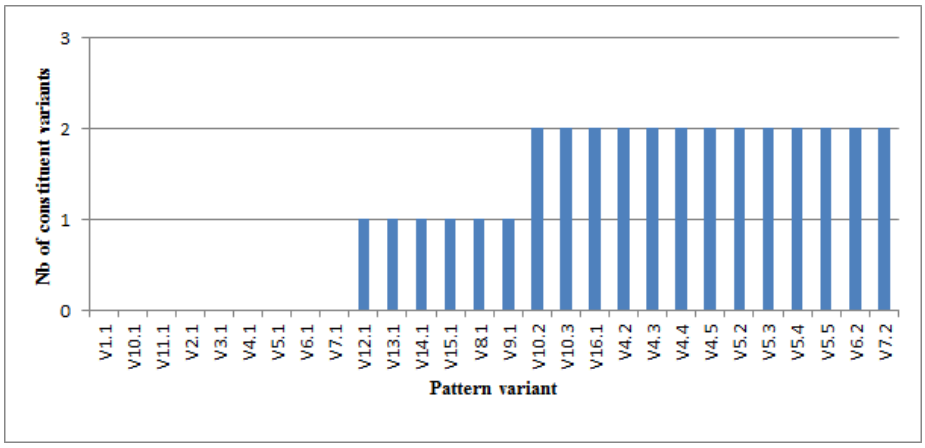


Figure 19: Most of pattern variants can be composed from other variants

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. 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.

### *7.3. 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 21. 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

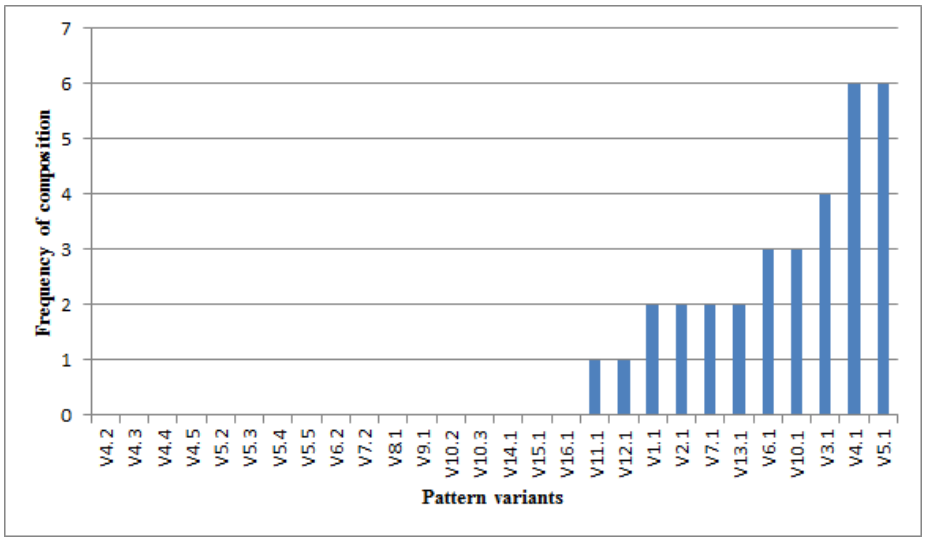


Figure 20: Frequency of composing a pattern variant

approach and measured how many other variants can be built from it. Figure 22 shows for each composed pattern, the number of possible reconstructions.

The variants chosen to participate in this evaluation are those created from the composition 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 22, 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.

### *7.4. 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.

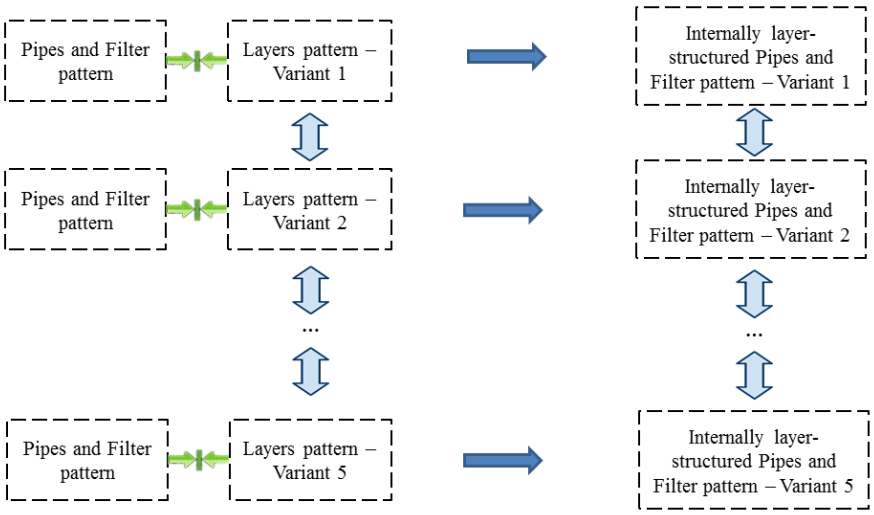


Figure 21: Reconstructability of composed pattern by switching between different variants of constituent patterns

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. However, 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.

### *7.5. 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,

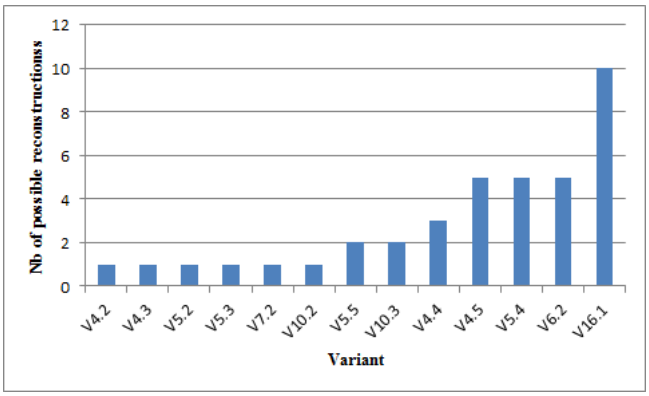


Figure 22: Frequency of composed pattern reconstruction by reusing merging operators

considering the flexibility and customizability of patterns <5>. Indeed, many patterns capture existing experience that are specific to certain projects, software systems or companies. However, the more patterns, the more crowded the variants and thus the more likely the approach has effect.

## 8. Conclusion

The use of patterns, when building architectures, has a twofold interest: the use of proven solutions to recurring problems, but also the support for documenting architectural choices. In one of our previous papers <24> we proposed a solution based on the use of patterns to handle the latter issue. Thus, this paper dealt with the former issue in the case of complex patterns.

Our proposition consists in 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 in case of changes and in this way allows the propagation of changes to the container 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 (bottom and top parts of Figure 10). 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. To validate this extension to support behavioural patterns, we plan to use as semantic foundation the $\pi$-ADL architecture description language <21>. This is motivated by the fact that $\pi$ADL is based on a comprehensive process calculus, the $\pi$-Calculus and subsuming other ADLs for expressing behavioural properties.

## References


[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 a pattern language. In *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee*, pages 1–39, 2005.

[3] 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.

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

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a system of patterns*. John Wiley & Sons, Inc., 1996.

[6] 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.

[7] 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.

[8] Jing Dong. Representing the applications and compositions of design patterns in uml. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 1092–1098. ACM, 2003.

[9] 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.

[10] 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, New York, NY, USA, 2000.

[11] 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.

[12] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *IEEE Softw.*, 24:38–45, 2007.

[13] ISO/IEC 25010:2011. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. ISO, Geneva, Switzerland.

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

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

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

[17] 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.

[18] 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.

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

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

[21] Flavio Oquendo. pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[22] Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. Ontology-based modelling of architectural styles. *Inf. Softw. Technol.*, pages 1739–1749, 2009.

[23] 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.

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

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

[26] 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. ACM, 2011.

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

[28] 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*, SHARK '08, pages 11–18, New York, NY, USA, 2008. ACM.

[29] Peng Liang Zengyang Li and Paris Avgeriou. Application of knowledge-based approaches in software architecture: A systematic mapping study. *Information and Software Technology*, 55(5):777 – 794, 2013.

[30] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann. Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. In *WICSA '08, 7th Working IEEE/IFIP Conf. on Soft. Architecture*, pages 157–166, 2008.