Supporting architectural pattern language using structural properties

Minh Tu Ton That, Salah Sadou, and Flavio Oquendo

Université de Bretagne Sud
IRISA
Vannes, France
{minh-tu.ton-that,Salah.Sadou,Flavio.Oquendo}@irisa.fr

Abstract. Architectural patterns(styles) are important artifacts containing specialized design knowledge to build good-quality systems. Complex systems often exhibit several architectural patterns in their design which leads to the need of architectural pattern composition. Unfortunately, information about the composition of patterns tend to be vaporized right after the composition process which causes problems of traceability and reconstructability of patterns.
This paper proposes a pattern description language that first, facilitates several types of pattern merging operation and second, allows the traceability of pattern composition. More specifically, the approach consists of a proper description of pattern that supports composition operations and a two-step pattern design process that helps preserve pattern composition information.

1 Introduction

Problems:
1. Patterns exist in complex forms which require the combination and reuse of other patterns. In the literature, the supports for pattern composition consist of using merging operators that are different from the pattern language which prevent the reusability as well as the composability of patterns.
Main contributions:
1. Give patterns and merging operators first-class status.
2. Support the design of hierarchical patterns

2 Problem statement

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. 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.

2.0.1 Blend of patterns By observing the documented patterns in [6,7], 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.

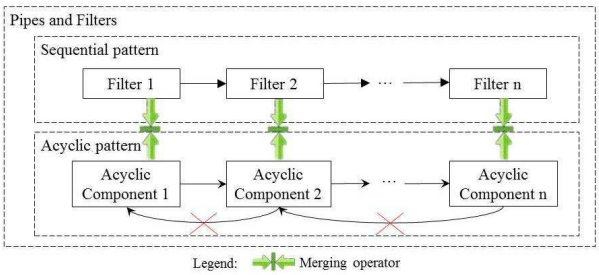

Fig. 1. Pipes and Filters

If we consider this type of structure as a sub-pattern, we can say that the pattern *Pipes and Filters* is composed of two sub-patterns (see Figure 1): The first called *Sequential pattern* consists of *Filter* components linked together by *Pipe* connectors and the second called *Acyclic pattern* 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.*

2.0.2 Connection of patterns A lot of documented patterns formed from two different patterns can be found in [7,2]. For instance, 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.

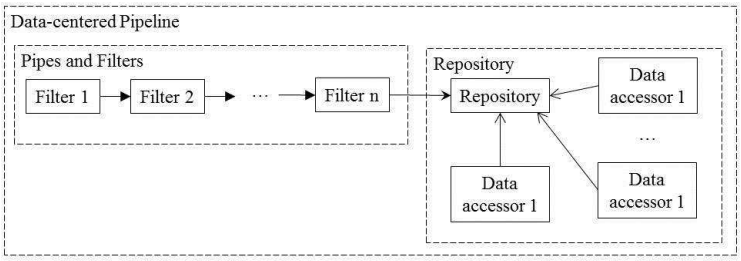

Fig. 2. The Data-centered pipeline pattern

As we can observe, the two patterns are linked together by a special connector which serve two purposes at the same time: convey data from a Filter and access to the Repository. But once the composed pattern built, it is difficult to identify the sub-patterns used in its constituent patterns.

2.0.3 Inclusion of patterns 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 3. 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.

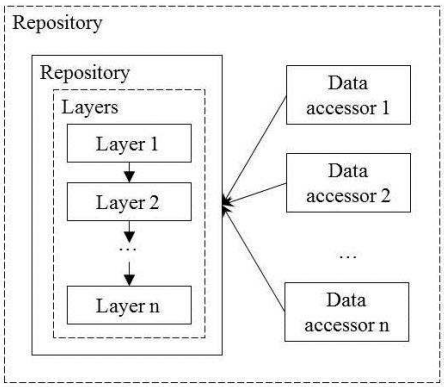

Fig. 3. Layers as internal structure of Repository

Despite the existence of this type of composition, the proposed works have not given the support for it.

2.1 Discussion

As we can observe from the example of sub-section 2.0.2, 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 2.0.1. 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, knowing 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 sub-section 2.0.1, 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 2.0.2 must be also reconstructed.

In the literature, the already proposed approaches such as [12,3,8] 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 totally ignore two aforementioned issues.

To realize the inclusion merging operation like one presented in the example from sub-section 2.0.3, the pattern description language should provide the recursive definition for pattern. More specifically, when specifying an element of a pattern we should be able to add other patterns inside to characterize the element's internal structure. To our knowledge, the proposed pattern languages have not give the optimal solution to this type of hierarchical composition.

In summary, the examples shown above highlight three 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*: Anytime 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.
3. *Support for hierarchical pattern composition*: While constructing a pattern, one should be able to build the internal structure of an element by including another pattern.

3 General Approach

We propose the process of constructing patterns including two steps as illustrated in Figure 4. The first step consists of describing a pattern as a composition graph of unit patterns. Thus, the pattern comprises many blocks, each block represents a unit patterns, all linked together by merging operators.

The second step consists of refining the composed pattern in the previous step by concretizing the merging operators. More specifically, depending on which 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 [17]. Each pattern is considered as a model conforming to its meta-model 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.

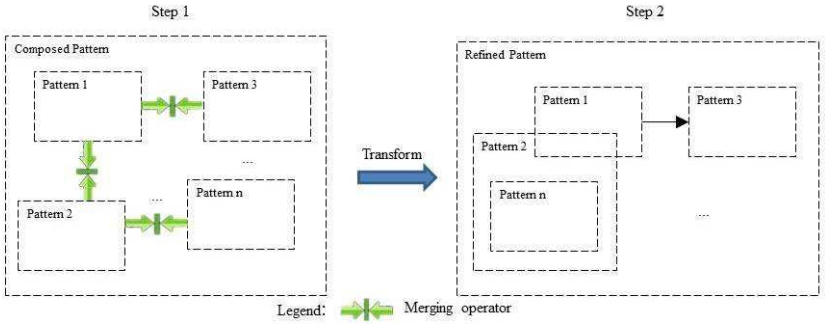

Fig. 4. Overall Approach

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 easily 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 thank 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.

Finally, to solve the third problem (support for hierarchical pattern composition), we propose to give pattern itself first-class status in our pattern description language. That means that patterns should play the same role as other elements where we can make connection with, add properties and most importantly, set them up as internal elements. This recursive definition of pattern gives the pattern description language the capacity to describe hierarchical patterns as mentioned in the illustrative example of Section 2.0.3.

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

4 A pattern language for hierarchical pattern and composition

We introduce our language called COMLAN (Composition-Centered Architectural Pattern Language) 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 [21] which serves at documenting architectural decisions about the application of architectural patterns. As shown in Figure 5, our meta-model is composed of two parts: the structural part and the pattern part. As pointed out in [14,1] and also described in [7], the design vocabulary of an architectural pattern (style) 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:

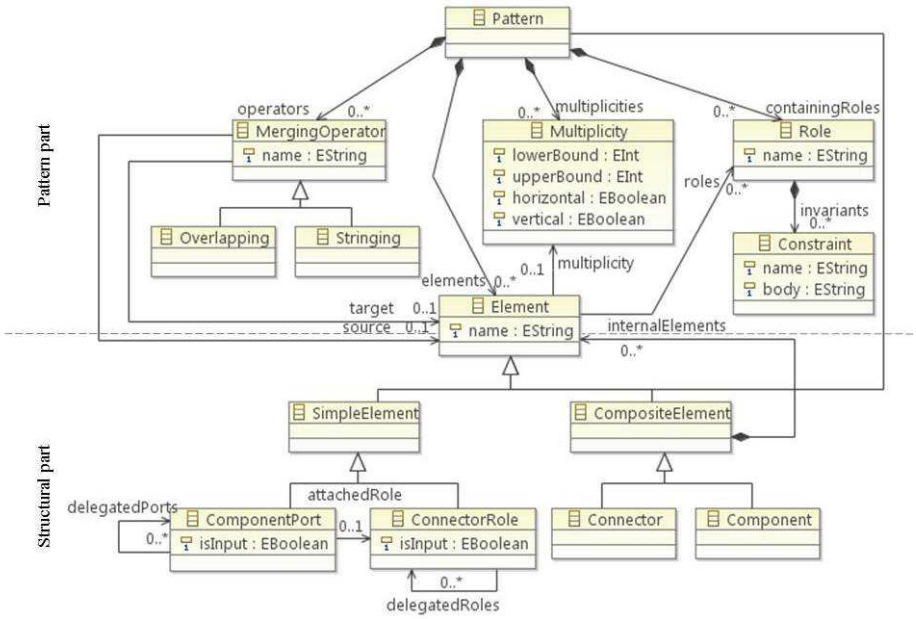

Fig. 5. The COMLAN meta-model

- *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
- *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.
- *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 5) 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 6 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 [6].

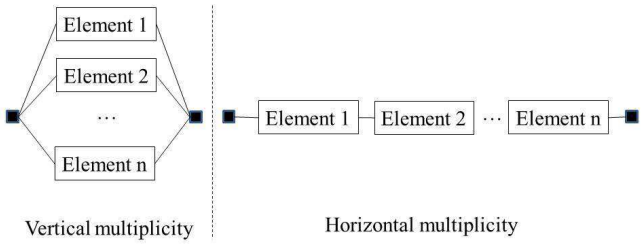

Fig. 6. Orientation organization of generic elements

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 on 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 [18] rules.
Similar to [4,12], in our language two types of merging operator are supported: stringing and overlapping as shown in Figure 7. 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*.

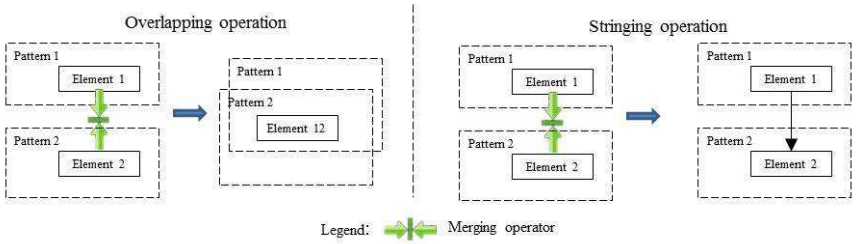

Fig. 7. Two types of merging operation

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 example

For the purpose of illustration, our pattern language will be used to model an example about the pattern for data exploration and visualization as in the Vistrails application's architecture in [5]. More specifically, this model represents the first step of the two-step pattern definition process. As shown in Figure 8, this pattern consists of three main sub-patterns: Pipes and Filters, Client-Server and Repository, all connected together through merging operators. Among these three patterns, the Repository pattern is a hierarchical one whose the component Repository 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 8, 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 *ConnectedFilter* constraint. To be more specific, it stipulates that a filter cannot stand alone, there must be at least one pipe

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

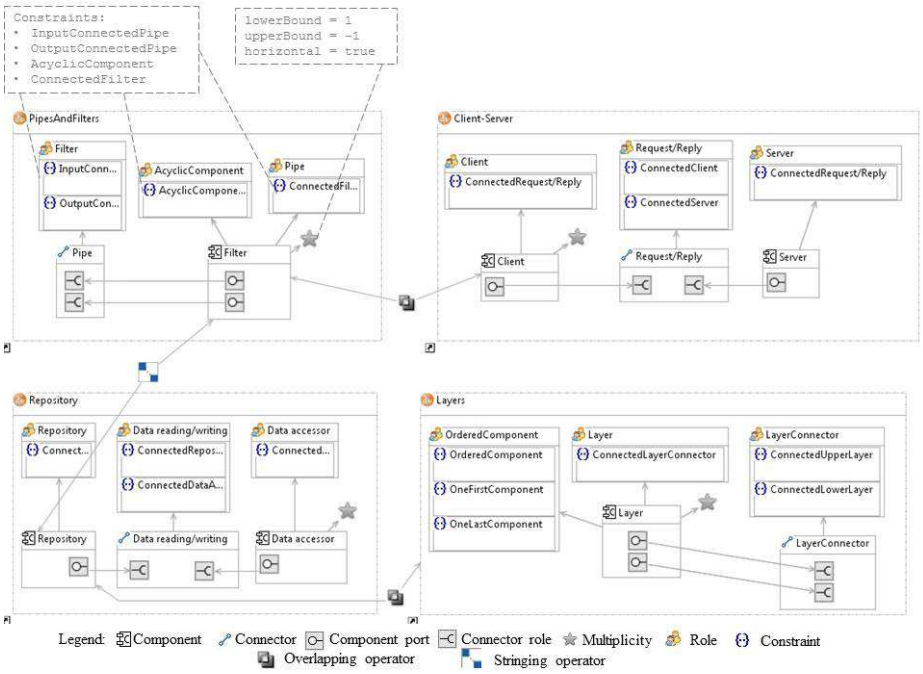

Fig. 8. Example of pattern model

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
   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 8), three merging operators are used: first, 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, second, 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 and finally, 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.

5 Pattern refinement

After being described as the composition of constituent patterns through merging operators, the pattern model will be refined. We consider this second step in the two-step pattern definition process as a model transformation from a pattern model where merging operators are explicitly presented to a pattern model where merging operators are concretized. 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 9, 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.

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.

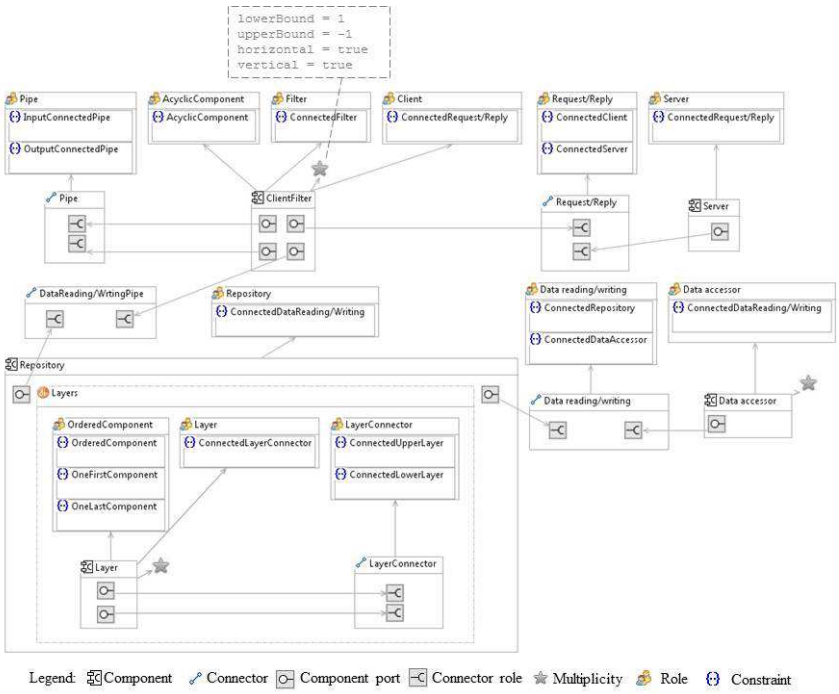

Fig. 9. The refined pattern model

For composite elements, the composition begins with the fusion of all internal elements. As we can see from Figure 9, 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* and *Filter* since it participates as a Filter in the Pipes and Filters pattern and *Client* since it participates as a Client in the Client-Server pattern.

The multiplicity is merged as follows: The lower bound of the merged element's multiplicity is the maximum of the lower bound of the source element's multiplicity and the lower bound of the target element's multiplicity. On the contrary, the upper bound of the merged element's multiplicity is the minimum of the upper bound of the source element's multiplicity and the upper bound of the target element's multiplicity. 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 9), the multiplicity of the

merged component *ClientFilter* is both vertical and horizontal as illustrated in Figure 10.

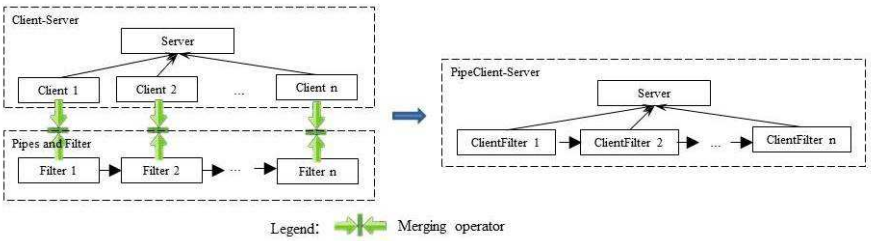

Fig. 10. The merged pattern of Client-Server and Pipes and Filters

In the case of a chain of consecutive overlapping operators in which one continues another, we use a special algorithm which is sketched in Figure 11. 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 2* and *Element 1* to create *Element 12*. Next, *Element 2* is replaced by *Element 12*. In the second step, the overlapping operator merges *Element 3* and *Element 2*, which is actually already replaced by *Element 12*, 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 thank 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.

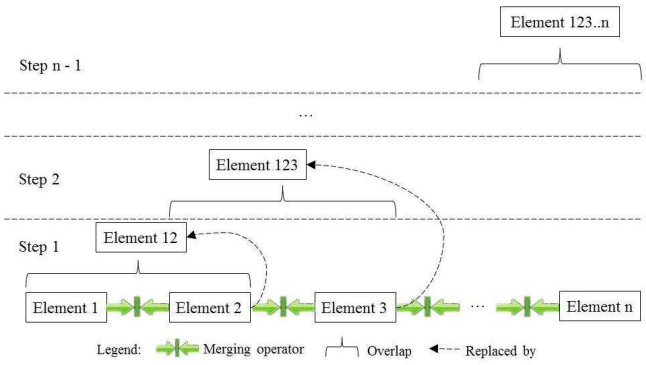

Fig. 11. The algorithm in case of multiple overlapping operators

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 9, 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 keep all its internal elements.

6 Implementation

To verify the feasibility of our approach, we developed the *COMLAN* tool, then we applied it to the case of Vistrails's architecture [5]. With *COMLAN* we aim to make concrete the aforementioned concepts. The tool provides the following functionalities:

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

COMLAN is developed based on EMF (Eclipse Modelling Framework) [20]. We choose EMF to realize our tool since we leverage MDA, where models are basic building units, to develop our approach. The tool consists of two Eclipse plug-ins built on existing Eclipse technologies. They are:

- *Pattern creation plug-in* uses EMF and GMF (Graphical Modeling Framework)[2] modeling support in order to allow architects to define *Pattern models* graphically. More specifically, the editor allows to design constituent patterns and compose them using two types of merging operators: stringing and overlapping. Furthermore, hierarchical pattern description is also supported. Besides, the editor allows the automatic propagation of changes in the constituent patterns to the already composed patterns. This editor depends on the *COMLAN Meta-Model* (see figure 5).
- *Pattern refinement plug-in* uses Kermeta [16] to implement rules transforming composed pattern model to refined pattern model. This functionality allows the architect to obtain a pattern with all the merging operators concretized, ready to be instantiated in the architectural model.

For a complete tutorial and a video about this tool, the reader is invited to visit the following website: `http://`

[2] Eclipse Consortium. "Eclipse Graphical Modeling Framework (GMF)(2007)."

7 Related work

Our work concerns three areas of related work. The first is architectural pattern (style) description language. In the literature there have been some efforts to model architectural patterns and their properties. For instance, there are works focusing on the use of formal approach to specify patterns. In the Acme ADL(Architectural Description Language) [10], the authors tend to provide a pattern-oriented architectural design environment where patterns are formally described. [19] uses an ontological approach for architectural pattern modeling based on a description logic language. As opposed to these domain specific languages, [15] proposes to use general purpose languages such as UML to model architectural patterns. Similarly, [11] suggests a mapping to transform from an ADL to UML 2.x to facilitate the use of UML in architectural pattern modeling. Applying a role-based pattern modeling approach, our language is designed to focus specifically on software architectural patterns. However, the genericness of the language is also assured since the pattern concepts used are those synthesized from many different ADLs.

The second area of concerned research is pattern composition. There are mainly two branches of work on the composition of pattern. The first including [12,3] proposes to combine patterns at the pattern level which means that patterns are composed before being initialized in the architectural model. On the contrary, the second including [8] proposes to compose pattern at instance level where an architectural entity is allowed to belong to different patterns. However, all of these approaches consider the composition as transient operation which leads to the problems we pointed out in previous sections. By proposing to give composition operators first-class status, our approach helps to prevent these shortcomings. In another work, [13] proposes a UML profile to attach pattern-related information on merged elements in composed patterns. With this approach, although one can trace back the constituent pattern in which an element participates in, a composition view showing how the original pattern is composed is still missing. Our proposal should also be compared with works on architectural constraint composition such as [22]. In this work, a pattern can be generally imposed by a constraint and complex patterns can be expressed through the composition of constraints. With our approach we raise the level of abstraction by using model to describe architectural patterns. Thus, not only the conformance of architectural patterns is assured but the application of patterns is also encouraged.

Finally, also related to our research is work on describing hierarchical pattern composition. In [23], the authors propose to use a number of architectural primitives to model architectural patterns. Through the stereotype extension mechanism of UML, one can define primitives (which equivalent to sub-patterns in our approach) to design a specific element of a pattern. However, the fact that pattern itself is not considered as an element in the pattern construction totally prevents its reusability. In our proposed pattern language, pattern is treated as first-class status which allows not only the modeling of primitives as patterns but also the reusibility of patterns to construct more coarse-grained patterns.

8 Conclusion

Summarize the importance of combinative pattern and hierarchical pattern in documenting patterns and the proposed approach.

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. Paolo Bottoni, Esther Guerra, and Juan de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Inf. Softw. Technol.*, pages 821–844, 2010.
5. Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, 2011.
6. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996.
7. 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.
8. 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.
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 Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188. ACM, 1994.
11. Simon Giesecke, Matthias Rohr, Florian Marwede, and Wilhelm Hasselbring. A style-based architecture modelling approach for uml 2 component diagrams. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, pages 530–538. ACTA Press, 2007.
12. 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.
13. Dong Jing, Yang Sheng, and Zhang Kang. Visualizing design patterns in their applications and compositions. *Software Engineering, IEEE Transactions on*, pages 433–453, 2007.
14. Jung Soo Kim and David Garlan. Analyzing architectural styles. *J. Syst. Softw.*, pages 1216–1235, 2010.
15. 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.

16. 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.
17. O.M.G. Model-driven architecture. `http://wwww.omg.org/mda`.
18. OMG. Object Constraint Language, OCL Version 2.0, formal/2006-05-01. Technical report, OMG, 2006.
19. Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. Ontology-based modelling of architectural styles. *Inf. Softw. Technol.*, pages 1739–1749, 2009.
20. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.
21. Minh Tu Ton That, S. Sadou, and F. Oquendo. Using architectural patterns to define architectural decisions. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP*, pages 196 –200, 2012.
22. 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.
23. Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Inf. Softw. Technol.*, pages 1003–1034, 2008.