Using Architectural Patterns to Define Architectural Decisions

Tu Minh Ton That, Salah Sadou, Flavio Oquendo

Using Architectural Patterns to Define Architectural Decisions

Minh Tu Ton That
IRISA
Université de Bretagne Sud
Vannes, France
minh-tu.ton-that@univ-ubs.fr

Salah Sadou
IRISA
Université de Bretagne Sud
Vannes, France
Salah.Sadou@univ-ubs.fr

Flavio Oquendo
IRISA
Université de Bretagne Sud
Vannes, France
Flavio.Oquendo@univ-ubs.fr

***Abstract*—During the architecture development process, architectural design decisions play an important role in maintaining non-functional properties of the system. Instead of supposing that architectural decisions are implicitly recognizable, existing works propose to give them first-class status. However, little focus is paid on the automation of architectural decision checking. This paper proposes to leverage pattern formalization techniques to document architectural decisions. The approach consists of a way to describe architectural patterns that hold the architectural decision definition, show how to integrate architectural decisions (patterns) into an architectural model and finally automate the architectural decision conformance checking.**

***Keywords*-architectural decision; pattern; SOA**

I. Introduction

One of the major problems of software development lies in the "maintenance and evolution" stage. Indeed, given the high costs associated with this stage (about 80% of the total cost), it becomes important to find a solution to reduce them. The main factors of this problem are the non-compliance with established practices and the lack of explicitness of the choices made throughout the development process.

If we are in the first case and we need to apply an evolution, we must first rebuild what has already been improperly built. Applying an evolution on a poorly constructed system can only make it more complex and ultimately not able to evolve further (Lehman second law [1]). In the second case, the system is well developed, except that the intentions behind each choice are not explicit. There are always different solutions to achieve a change, but some of them may be in contradiction with certain implicit intentions. Moreover, it can take several steps between the creation of the contradiction and its detection. This requires undoing what has already been built, resulting in an important additional costs.

The first factor of the problem cited above may be avoided by human or automatic controls [2] to check compliance with good practices. To avoid the second factor, we need to make explicit and exploitable intentions that lie behind each choice. The explicitness must begin at the software architecture definition stage. In this paper we will focus on this last point.

The intention associated with an architectural choice is designated in the literature by the term "Architectural Decision" (AD) [3]. Thus, the objective is to define the links that bind the components of AD: the property identification, the involved architectural elements and the rules defining the property. A formalization of this link serves to automatically check that an evolution does not conflict with the choices already made. Several studies have already been made to define such links [4], [5], [6], [7]. Proposed solutions usually consist of elements added to the architecture (constraints, specification, etc.) to establish the link. Although these elements indicate the presence of ADs, they do not encourage the architect to use the best solutions and/or good practices. Moreover, the added elements are often described using a language that is different from the ADL used for the architecture description. So, understanding the architecture requires a review of different elements from different languages, which complicates the task. As understanding the architecture is a step prior to its evolution, its complication undoubtedly induces a significant cost.

With this paper we propose to make the architectural decision a first class entity in the architecture description languages. We chose to use the architectural patterns as a support for describing the architectural decisions. In our approach, an architecture consists of architectural elements, their relationships and a set of definitions of architectural views. From a given architecture several architectural views can be obtained, but only a few among these views really have meaning. The views that we associate with the architecture are those that highlight the architectural decisions. Thus, when a view conforms with a pattern, it implies the presence of an architectural decision.

The remaining of the paper is organized as follows: Sect. 2 introduces the general approach, Sect. 3 goes into detail the reusable AD creation step, Sect. 4 describes AD manipulation stage, Sect. 5 introduces the implemented tool and the experimentation, Sect. 6 discusses related work, and Sect. 7 concludes the paper.

II. General Approach

The main idea behind our work is the leverage of architectural patterns as forms of AD representation. Similar to [8], AD documentation in our approach falls into three steps:

AD creation, AD integration and AD verification. *Decision creation* consists in the specification of an AD made to an architectural model. A decision could be specific to a project or reusable within different projects. Architectural decisions could be well-known architectural patterns which are lessons learned from many previous works or decisions that have high potential to be reused in an enterprise. *AD integration* is a step in which architects link ADs with affected elements in the architectural model. During the *AD verification* step, the architectural model is checked whether it complies with the integrated ADs.

On the purpose of automating the process of AD documentation, we use the Model Driven Architecture (MDA) approach [9]. Each artifact is considered as a model conforming to its meta-model in order to create a systematic process thanks to model transformations and leverage existing MDA techniques (e.g. conformity verification).

In the remainder of this section, we will go further into each step in the AD documentation process, clarify its objectives and explain how we achieve them with our approach. To be more concrete, in the remainder we will use the case of service-oriented architectures.

A. Decision creation

The first objective of decision creation is to facilitate communication between individuals and teams by creating a visual, compact AD model. In a collaborative working environment, ADs are often shared among members and teams in the project. Therefore, compact, visual AD models are welcomed instead of immense and complex ones. Our second objective is to increase reusability by language-independent ADs. During the process of reusing ADs, it becomes apparent that architects are likely to deal with architectural models described in different languages. As a consequence, a language-independent AD documentation model supporting adaptation to whatever language is needed.

Our approach relies on the fact that an architectural decision can be represented generically by a pattern. As shown in Figure 1 (Pattern definition part), we propose the use of a *general pattern meta-model* which contains only architectural elements involved in the AD definition. These elements are determined through a survey of well-known architectural patterns [10] according to the used ADL and some properties related to the way to define patterns. Thus, this meta-model is dependent on the family of the used ADL. In this paper, we propose a general pattern meta-model for the Service-Oriented ADL family that allows description of patterns such those defined in [11]. Based on this *general pattern meta-model*, one can define a meaningful architectural pattern in form of a *pattern model* using only necessary elements and hence, the first objective is satisfied. Furthermore, *pattern models* are also language-independent. With the separation between AD documentation and architectural design, no modification to the architectural model is needed to define an AD, which makes it easy to adapt to different architectural languages of the same family.

B. Decision integration

Links between AD elements and their correspondent architectural elements play an important role in keeping track of AD made to an architectural model. An explicit linking will facilitate the AD documentation as well as the AD modification (if there is one) in the future. In our approach, links between AD elements and architectural elements are represented by *mapping models* (illustrated in the Pattern verification part of Figure 1). We can observe here that an architectural model is now a combination of architectural elements and *mapping models* associated with them. A *mapping model* indicates that an AD is made on an architectural model.

In the literature, architecture is considered as a set of views which are representations of system elements and relations associated with them [12]. Each view serves a specific purpose depending on the concerns of one or more stakeholders. Having taken this viewpoint into account, we propose to consider an architectural model as a multi-view representation where each view contains only elements related to a specific AD. In this scenario, *architectural views* are filtered from the architectural model through a transformation mechanism in which *mapping models* play the role of integrating AD's information into the architectural model.

C. Decision verification

To make sure that an architectural model is consistent with ADs, not only do the existence of AD-related elements in an architectural model need to be verified but also the constraints imposed on them need to be handled. To achieve the first goal, the presence of ADs in the architectural model is checked through the completeness of *mapping models*. Indeed, *mapping models* are intermediary bridges between the architectural model and ADs and thus, the incompleteness of *mapping models* shows the lack of ADs in the architectural model. To achieve the second goal, the constraints imposed by ADs on the architectural model are checked through the conformity of *AD architectural views* with their corresponding *AD view models*. To check the conformity of views to patterns it is possible to define similarity functions between models, as it was done in [13] or leverage the meta-model representation form of patterns as in [14]. We chose to first transform the architectural patterns into meta-models (AD view meta-models in figure 1) in order to make use of consistency checking tools from MDA. Indeed, we believe that defining a function that checks the conformity of a view (model) with a pattern (generic model) would lead largely to redefine a function which checks conformity of a model with a meta-model. And contrary to [14], we do not modify the initial language

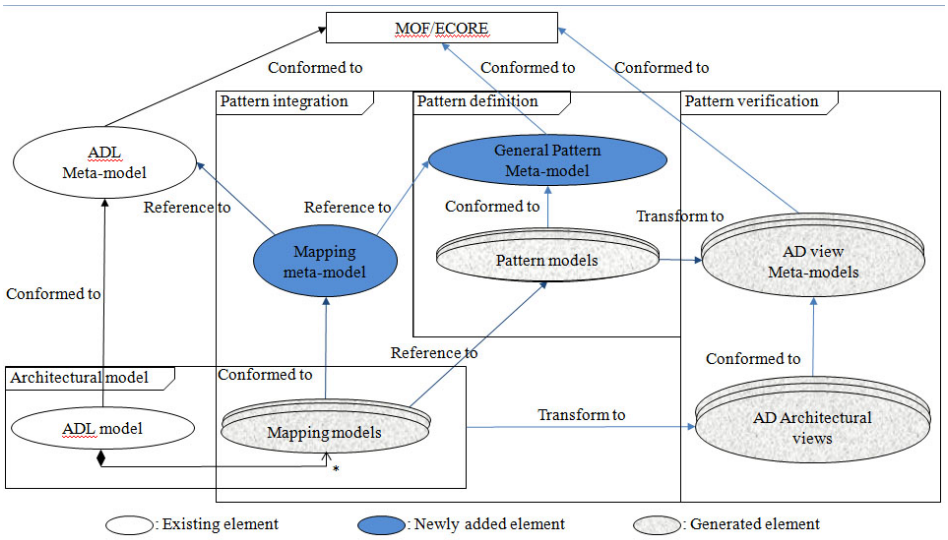

Figure 1. MDA approach for AD documentation

while we generate the necessary meta-models (one by used architectural pattern) just for checking ADs.

III. Reusable AD creation

After having generally introduced the reusable AD creation stage in the previous section, in this section we aim at going further into detail. As we choose architectural pattern as the form of AD representation, the process of creating an AD consists in specifying a *pattern model*. We first introduce the *General pattern meta-model* from which ADs are created. Next, we clarify the AD creation process through a concrete example.

A. General pattern meta-model

The *general pattern meta-model* provides the language to define an architectural pattern. It contains all necessary architectural features from an ADL to create a pattern. Since we validate our approach in the SOA domain, our *general pattern meta-model* is purposely proposed for SOA. As shown in Figure 2, the meta-model is composed of 2 parts: pattern structure and architectural structure.

Inspired by the SCA model[1] [15], we construct the structural aspect of our *General Pattern Meta-Model* for the SOA description language family as follows:

- *Composite* serves as the container to assemble and connect service-oriented building blocks together.
- *Components* are basic units of the architecture that represent business functions from which composite applications are built.
- A component is composed of *component services* and *component references*. The former provide functionalities supported by the component and the latter play the role of consuming services of other components. A component reference can be wired to a component service through its *target* attribute.
- Thinking of composites as black-box components, they have also *services* and *references*. To be consumed by the world outside of a composite, a component service of the containing component can be promoted as a service of the composite. Similarly, to be served by an outside service, a component reference should be promoted as a composites reference.

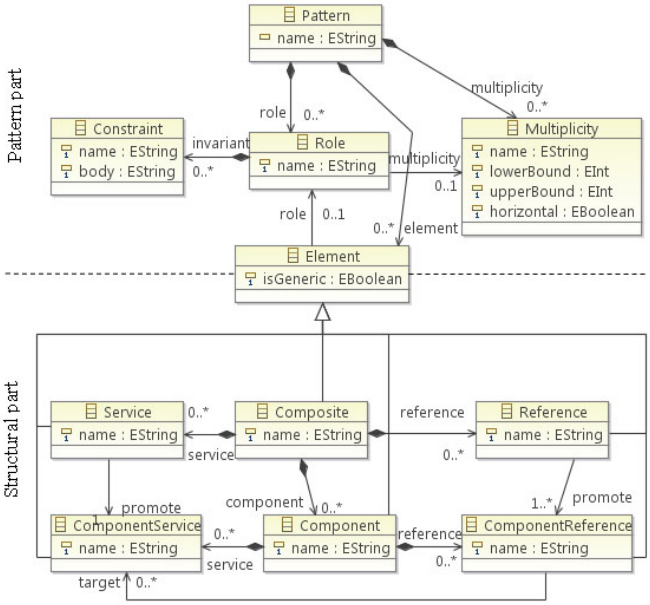

Figure 2. SOA General Pattern Meta-model

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

The *pattern aspect* part 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 at two level: generic and concrete. Via the *isGeneric* attribute, we can specify an element as generic or concrete. A concrete element provides guidance on a specific pattern-related feature. Being generic, an element represents a set of concrete elements playing the same role in the architecture. 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 [14]. To characterize a role, we provide two supports: *constraint* and *multiplicity*. A constraint made to a role on an element helps make sure that the element participating in a pattern has the aimed characteristics. Constraints are represented in our approach in form of OCL [16] rules. A multiplicity indicates *how many times* a pattern-related element should be repeated and *how* it is repeated.

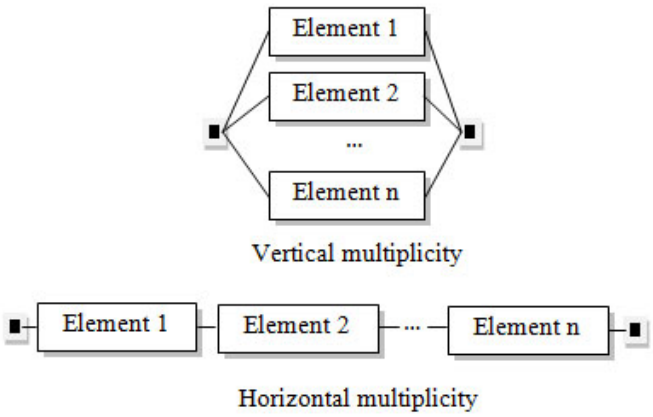

Figure 3. Orientation organization of generic elements

For instance, we can specify that an element in a pattern should be repeated in a maximum of three times and the repeating instances of this element should be organized horizontally. Figure 3 shows two types of orientation organization: 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.

B. Architectural Pattern Specification

As described in the previous section, in our approach a reusable AD is represented as a fully understandable architectural pattern with cohesive elements and constraints imposed on them. For the purpose of illustration, we will examine the SOA Legacy Wrapper pattern [11].

As shown in Figure 4, the SOA Legacy Wrapper pattern stipulates that in order to eliminate legacy technical details

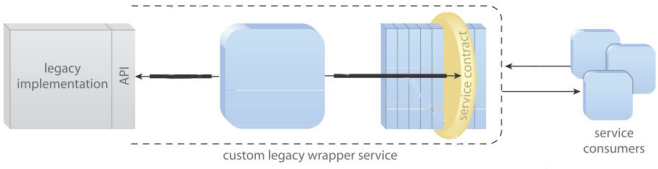

Figure 4. SOA legacy wrapper pattern [11]

from a legacy component, one should use a wrapper service equipped with a standardized service contract. In other words, every service in the architecture must communicate with the legacy component (if needed) through the legacy wrapper bridge to avoid the potential technical incompatibility. This architectural pattern aims at assuring the system interoperability.

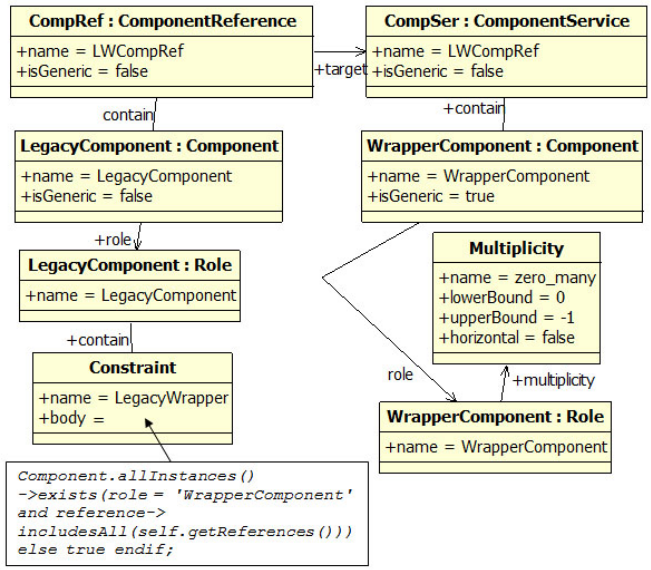

Figure 5. SOA Legacy Wrapper pattern model

Based on the *general pattern meta-model*, we can specify the *pattern model* for the SOA Legacy Wrapper with the emphasis on the following elements (as illustrated in Figure 5): the component *LegacyComponent* specified with the role *LegacyComponent* representing the component with legacy implementations, the component *WrapperComponent* specified with the role *WrapperComponent* representing the wrapper services in the pattern. The component *LegacyComponent* has the *isGeneric* attribute assigned to false since it represents a concrete legacy component. Otherwise, the component *WrapperComponent* has the *isGeneric* attribute assigned to true since it represents many possible wrapper components. The role *LegacyComponent* is characterized by the *LegacyWrapper* constraint. To be more specific, it stipulates that if a component has the *LegacyComponent*

role then there must exist a component with the *WrapperComponent* role in such a way that all services coming from the Legacy Component are referenced by the Wrapper Component. The role *WrapperComponent* is characterized by a multiplicity class specifying that there maybe many instances of WrapperComponent and moreover, they must be vertically connected. Even though the other participating elements such as the component service and the component reference do not have specific roles, they still contribute to the model to make a meaningful pattern.

IV. AD Manipulation

In our approach, the ADs are on one hand integrated into the architecture description to complete its documentation and on the other hand, they are used as a means to verify that the architecture is consistent with the taken decisions. The integration of an AD to an architectural model is made thanks to a mapping model, and the checking of the conformance of the architecture with an AD is made thanks to a particular view on the architectural.

A. Associating an AD to an Architectural Model

The association of an AD with an architectural model consists of defining a mapping model between the latter and an architectural pattern representing the AD. The mapping model is part of the architectural model as a means to associate an AD to the architecture. Concretely, it links elements in the architecture that directly relates to elements from the pattern model.

The mapping meta-model consists of one meta-class per type of mapping. So, each meta-class defines a mapping between an element's type from the ADL meta-model that may participate in architectural patterns and an element's type in the *general pattern meta-model*.

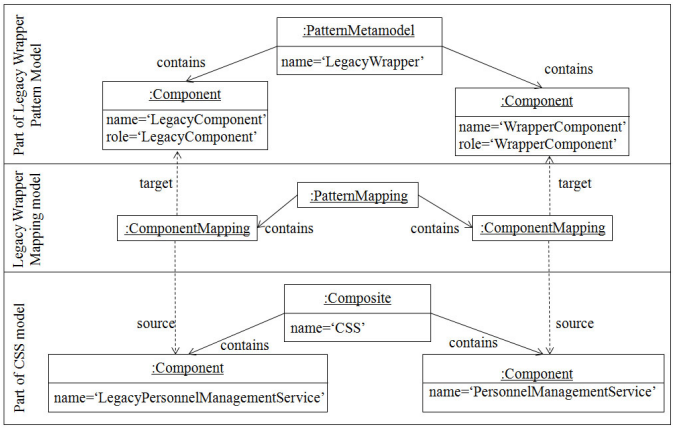

Figure 7. Mapping model for SOA Legacy Wrapper pattern in CSS

For the purpose of illustration, we built an architecture for the Client Support System (CSS) (as shown in Figure 6) which aims at supporting a company to receive, manage and answer client's requests systematically. Figure 7 sketches the mapping model which associates the Legacy Wrapper pattern to a part of the CSS architecture. As we can see in this figure, two components *LegacyPersonnelManagementService* and *PersonnelManagementService* in the CSS architecture are mapped respectively to two components playing the roles of legacy component and wrapper component in the Legacy Wrapper *pattern model*.

B. Extracting AD views

The architectural views are very useful for understanding the overall architecture of a complex system. In our case, a view is constructed by applying a transformation on the architectural model which targets a given AD. The transformation serves as a filter to realize two purposes: first, extract from the architectural model elements relating to ADs and second, eliminate language-specific features to create a language-independent architectural model. Therefore, this can be compared to the transformation step from PSM (platform specific model) to PIM (platform independent model) in the MDA approach. To realize this, we leverage the MDA transformation techniques. More precisely, we use ATL [17] transformation rules to transform architectural models into AD view models.

Thus, the transformation consists of:

- From the architectural model, transform one by one only elements whose equivalent type is specified in the *pattern model*.
- Based on the *mapping model*, copy role information of mapped elements from the architectural model to their equivalent elements in the AD view.

The following is an excerpt of one of the rules associated with the *SOA Legacy Wrapper pattern* used in the mapping of figure 7:

```
rule Composite2Composite {
 from
  scai : SCAMM!Composite(GeneralSOAMM!Composite.
                         allInstances()->notEmpty())
 to
  spo : SOAPatternMM!Composite(
   name <- scai.name,
   role <- if (not scai.getMappingFromComposite().
                         oclIsUndefined())
     then scai.getMappingFromComposite().target.role
     else OclUndefined endif,
   component <- if (GeneralSOAMM!Component.
                         allInstances()->notEmpty())
    then scai.component
    else OclUndefined endif,
   service <- if (GeneralSOAMM!Service.
                         allInstances()->notEmpty())
    then scai.service
    else OclUndefined endif,
   reference <- if (GeneralSOAMM!Reference.
                         allInstances()->notEmpty())
    then scai.reference
    else OclUndefined endif
  )
}
```

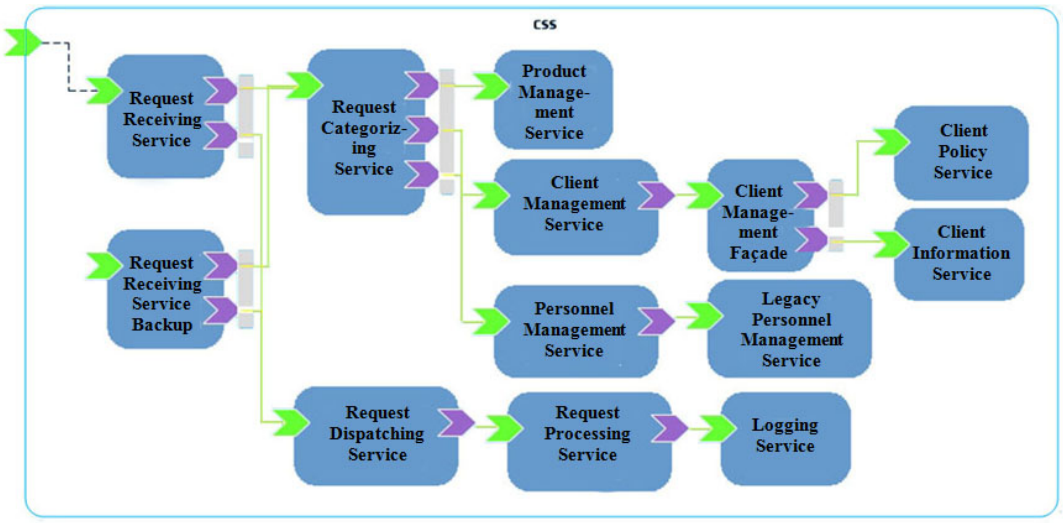

Figure 6. Client Support System architecture

As illustrated, the rule above transforms a SCA composite to a composite according to the AD view. First, the rule searches through the mapping model if there is a mapping in which the SCA composite is a source. If there is, the rule transforms the role the composite plays in the pattern to a composite in the AD view. Second, based on the existence of the component, the service and the reference in the pattern model, the rule transforms their values respectively to their corresponding in the AD view. The result of this transformation is an AD view which contains only AD-related elements with roles embedded on some of them.

C. AD Checking

The conformity of an architectural model with one of its associated ADs is verified through the conformity of the extracted AD view with the concerned architectural pattern. We chose to leverage the MDA verification technique to realize this step. Therefore, in our approach, AD view meta-models are generated from pattern models for the purpose of checking. The consistency of an AD view is thus verified against its corresponding AD view meta-model.

For every pattern model defined, an AD view meta-model is generated containing a subset of meta-classes, from the general pattern meta-model, that participate in the definition of the pattern model. Applying this principle to the Legacy Wrapper pattern model described in the previous section, we obtain an *AD view meta-model* with the participation of only three meta-classes: *Component*, *ComponentService* and *ComponentReference* as shown in Figure 8. This meta-model is embedded with invariants imposed on the Component meta-class as follows:

```
invariant legacyWrapper:
  if role = 'LegacyComponent'
     then Component.allInstances()->
           exists(role = 'WrapperComponent'
```

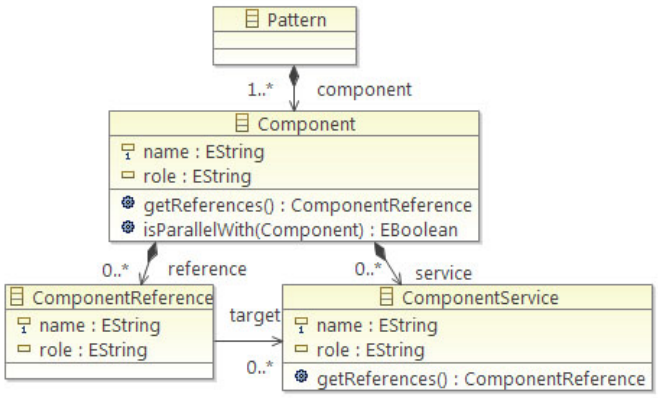

Figure 8. AD view meta-model for the Legacy Wrapper pattern

```
            and reference->includesAll
                (self.getReferences()))
     else true
  endif;

invariant orientation:
  if role = 'WrapperComponent'
     then Component.allInstances()->
           forAll(role = 'WrapperComponent' implies
             isParallelWith(self))
     else true
  endif;

invariant multiplicity:
  let s: Integer = Component.allInstances()->
       select(role = 'WrapperComponent').size()
       in s >= 0;
```

Invariants on meta-classes correspond to constraints specified on their instances that have a specified role in the pattern model. Furthermore, information about orientation and multiplicity are also reflected in the meta-model via invariants

on meta-classes. We can observe through the example that the constraint imposed on the *LegacyComponent* role in the pattern model is transformed into a *legacyWrapper* invariant on the *Component* meta-class with a condition on the *LegacyComponent* role. The multiplicity of the *Wrapper-Component* role in the pattern model is transformed into two other invariants in the *Component* meta-class: *multiplicity* and *orientation*. We will not present the transformation rule here because of the lack of space but basically it consists in:

- Transform one by one all elements with their role information.
- Add relations between elements based on their existence in the target meta-model.
- Constraints specified on role are reflected in the meta-class.
- Multiplicity information specified on roles are reflected in the meta-class.

After being generated, the AD view meta-model is used to check the the consistency of AD view. Whenever there is a violation of constraints imposed by the AD view meta-models on AD views, warnings are notified to the architect about which AD is violated and which elements in the architectural model are involved.

V. Implementation and Experimentation

To verify the feasibility of our approach, we developed a tool called *ADManager*, then we applied it to the case of SOA.

A. ADManager *tool*

With *ADManager* we aim to make concrete the aforementioned concepts. The tool provides the following functionalities:

1) Create architectural patterns (support of ADs definition)
2) Integrate ADs to architectural models
3) Verify the consistency of architectural models according to the held ADs.

ADManager is developed based on EMF (Eclipse Modelling Framework). We choose EMF to realize our tool since we leverage MDA, where models are basic building units, to develop our approach. As shown in Figure 9, the tool consists of five Eclipse plug-ins built on existing Eclipse technologies. They are:

- *Pattern creation plug-in* uses EMF modeling support in order to allow architects to define *Pattern models*. This editor depends on the *General Pattern Meta-Model* (see figure 1, which is given as a parameter. Thus, according to the targeted ADL family we need to give the corresponding *General Pattern Meta-Model*.
- *AD integration plug-in* is an editor supporting the creation of *Mapping models* between pattern elements and architectural model elements. It depends on the *Mapping Meta-Model*, which is also depending on the targeted ADL family.
- *AD verification plug-in* uses OCL tool to support writing rules in pattern models, during pattern creation, as well as conformance verification between AD view models and AD view meta-models during AD checking. It relies on *General Pattern Meta-Model* to check the consistency of OCL rules.
- *AD view meta-model generator plug-in* uses ATL to implement rules generating AD view meta-models from pattern models. It relies mainly on *General Pattern Meta-Model*.
- *AD view generator plug-in* uses ATL to implement rules generating AD views from architectural models. It relies mainly on the mapping models attached with the latter.

Thus, for *ADManager* to work, it needs to have the *General Pattern Meta-Model* and the *Mapping Meta-Model* that are already defined. We did it for the SOA description language family (see the previous section). Thus, to switch to another language family (such as object-oriented), we must provide the appropriate meta-models. The *Mapping Meta-Model* can be produced very easily when we have *General Pattern Meta-Model*. To produce the latter we need a good knowledge on the ADL family. Note that in Figure 2 the example of *General Pattern Meta-Model* for SOA is separated into two parts: one specific to the SOA description language family and the other for the notion of pattern. Thus, this example can be reused for producing a new *General Pattern Meta-Model* only by redefining the part related to the targeted ADL family.

B. Experimentation with SOA Patterns

As can be guessed from reading our paper, we experimented with our approach on Service-Oriented ADLs. Thus, we wanted to verify that our tool allows the definition of well known SOA patterns. So, we have examined the SOA patterns from [11] (Synthetized in table I).

In the table I we reused the categorization of patterns given in [11]. Among the 74 identified patterns there are up to 45 patterns focusing on the aspect of service management

Pattern category	Nb of patterns	Nb of architectural patterns	Nb of formalized patterns
Service inventory design pattern	20	10	0
Service design pattern	31	13	11
Service composition design pattern	23	6	2

Table I
Categories of SOA Patterns from [11]

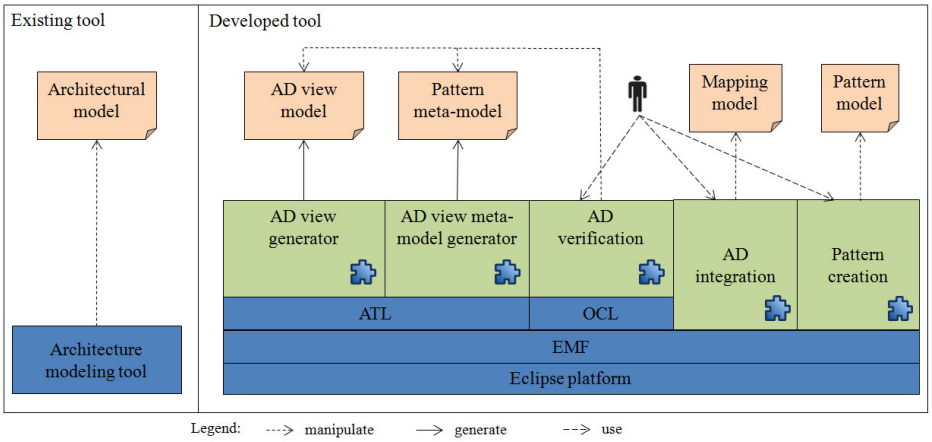

Figure 9. The architecture of ADManager

such as how to centralize or decentralize services physically, how to determine the boundary of service logic, how to add additional routines to service processing, etc. Therefore, these patterns cannot be documented using concepts from the ADL.

In fact, as we can observe in the table I, Among the remaining 29 architectural patterns there are ones based on architectural concepts that are not supported yet by SOADLs such as service inventory, service layer, service state data, data schema... That explains why only 13 patterns are documented using our approach. Most of the formalizable patterns fall into the Service design pattern category. Indeed, patterns in this category are good practices in service organization, encapsulation, implementation, governance and therefore, suitable to be documented architecturally. However, all those falling into this category are not related only on architectural aspects. For instance, the service identification patterns aim to conceptually decide a suitable level in which solution logic is decomposed and service encapsulation is identified are not documented. It is obvious that these patterns serve a purpose in organizing the logic of the system and thus, they are not purely architectural patterns.

in the Service composition design pattern category there are two patterns (namely Neutral Sub-controller and Brokered Authentication) that are possibly documented due to the fact that they manipulate certain architectural elements to stipulate their logic such as service, composite service, etc.

As illustrated in the designed CSS architecture (see figure 6), we documented the three following patterns: Legacy Wrapper, Service Façcade and Redundant Implementation. Except for the Legacy Wrapper pattern which is well explained as an example in previous sections, we can also observe the two other patterns in CSS: The Client Management Façcade component plays the role of a façcade for the two components Client Policy Service and Client Information Service; the Request Receiving Service component is backed up by the Request Receiving Service Backup component.

First, we have successfully checked that the architecture respects the three ADs. After that, we have made several modifications to the architecture where we know they lead to an AD violation. Each time the tool indicates the correct violated AD and which elements in the architecture are involved. Therefore, the test succeeded.

C. Discussion

The pattern description language is based on the target ADL as shown in Figure 2. If we take the meta-model of the target ADL as basis to define the pattern description language, we get a language capable of expressing all what can express the target ADL. Two goals are sought by allowing the description of patterns related to a project or a company: making architectural decisions explicit and reusable at least in their context (project or company). This last point implies a certain genericity in the description of patterns. We know that all elements of the target ADL are not necessary to describe patterns. So, ignoring them in the meta-model of pattern description allows for a simplified and more generic language.

Thus, we must find a balance between lightness and completeness of the pattern description language. For the case of SOA (see figure 2, which is used as example in this paper, we do not claim that we found the best balance. However, our experiment showed us that we can express the classical patterns related to SOA. We do not need to validate the coverage of all possible patterns for two reasons: i) It is not possible to imagine all possible architectural decisions. ii) the coverage depends on the meta-model of pattern description that is a data of our approach.

Therefore, the design of the meta-model for pattern description requires not only a good knowledge of the target ADL and its meta-model, but also extensive experience of its

use. Thus, it would be interesting to define meta-models of architectural pattern description for the classical ADLs and make them available to others. Sharing these meta-models allows on one hand to help others and on the other hand, to get feedback to improve them. For companies that use exotic ADLs, the definition of such a meta-model should be left to more experienced architects.

The reader may obtain a complete guiding tutorial video and more information about the *ADManager* tool using the CSS application at http://www-valoria.univ-ubs.fr/SE/ADManager.

VI. Related work

Our work directly concerns the definition of architectural decisions, but also concerns the AD conformance checking aspect. As we have chosen to represent ADs through architectural patterns, our work is also related to works on pattern definition. Thus, in the following we will discuss work related to these three aspects.

A. AD Documentation

In the literature there are many proposed models and tools supporting AD documentation. Among these works, we can mention some representative models such as the architectural decision template [18], the ontology of design decisions [19] or recently the MAD 2.0 model [20], and tools such as Archium [3], ADDSS [21], AREL [22]. We can observe that most of these works concentrate on capturing and characterizing ADs but none of them provide the automated checking of design decision compliance in architectural models. In our work, we do not attempt to just define another AD model but propose a way to define ADs which allows to automatically detect their violation in the architectural model.

In [8], Zimmermann et al. point out the importance of reusable ADs in decision identification, decision making and decision enforcement and propose a model to document reusable ADs. Furthermore, in [23], they propose to weave pattern languages into reusable architectural decision models to benefit their mutual interests. Besides that, in [24], Harrison et al. compare pattern and AD and think that the former can be leveraged to document the latter. We found these ideas interesting, thus we have gone further with our approach, which formalizes the representation of ADs with patterns.

B. AD Conformance Checking

Being one of the first works dealing with AD conformance checking, Tibermacine et al. [4], [6] propose a family of architectural constraint languages to describe the structural part of AD. Architectural constraints are used as a means to formalize ADs. With our approach we raise the level of abstraction by using architectural patterns to document ADs. ADs are no longer architectural constraints imposed on the architectural model but self-contained semantic pattern models.

In another work, Könemann et al. [5] propose a linking model to bind architectural decisions and architectural models. The consistence of the architectural model is checked through the completeness of the binding model. The structuring part of the AD is specified through model differences. In our approach, the conformance of the AD against its architectural model is checked through not only the mapping model but also the AD model itself. By using pattern to describe the structuring part of AD, we make sure that an AD is reflected semantically in the architectural model.

C. Pattern Definition

In [14], France et al. modify UML to incorporate pattern definition features. As architectural decisions are generally specific to concrete projects, patterns in our approach must not be static in a language and thus, the support of defining new patterns becomes essential. In [25], Elaasar et al. propose to bring concepts from the pattern specification language to the meta-meta-model. The modified meta-meta-model called Epattern is used to specify pattern meta-models. We can consider this approach as a pass from meta-meta-model level to meta-model level to define pattern language. In another work [13] et al. propose to express patterns as model snippets. The conformity of patterns is checked through pattern-matching functions between pattern models. We consider this approach as a pass from model level to model level to define pattern language. As opposed to the two above approaches, our approach defines patterns as models and transform them to meta-models for verification and thus, can be considered as a pass from model level to meta-model level to define pattern language. Thus, we can reuse the classical features concerning the conformity of a model to a meta-model.

VII. Conclusion

Keeping track of ADs made on a system is very important to avoid degrading it during its evolution. Although there existed a lot of works focusing on documenting ADs, the automation of AD checking during the development of the architecture is still an open issue.

In this paper, we propose to document ADs in the form of formalized patterns. This approach helps guarantee the existence of ADs not only in syntactic aspect but also in some semantic aspects. With the presented approach, the purpose of AD checking, which is an error-prone task, was automated. Besides, the reusability characteristic of AD is also taken into consideration since we leverage a language-independent AD creation mechanism. We also implemented a tool to realize our approach. We utilize the case of SOA to validate our approach but it is thoroughly relevant to other types of ADLs.

Still, we recognize some limitations in our approach. For example, considering the complexity of possible architectural patterns, we think that our general pattern meta-model is still not entirely sufficient. For instance, our pattern model only supports the multiplicity specification at element level, which makes it difficult to specify complicated patterns. For instance, when a pattern is composed of other patterns we may need to specify a multiplicity on some of the latter.

To overcome the above limitation, we are currently working on the way to incorporate more supports in the general pattern meta-model. For instance, to increase the generality of the multiplicity attribute of a role in the pattern, we intend to make it attachable to a group of elements. Thus, our pattern becomes a composite pattern composed of internal sub-patterns.

REFERENCES

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, pp. 1060–1076, 1980.

[2] V. L. Gloahec, R. Fleurquin, and S. Sadou, "Good architecture = good (adl + practices)," in *QoSA*, 2010, pp. 167–182.

[3] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, 2005, pp. 109–120.

[4] C. Tibermacine, R. Fleurquin, and S. Sadou, "Preserving architectural choices throughout the component-based software development process," in *WICSA*, 2005, pp. 121–130.

[5] P. Knemann and O. Zimmermann, "Linking design decisions to design models in model-based software development," in *Proceedings of the 4th European conference on Software architecture*. Springer-Verlag, 2010, p. 246262.

[6] C. Tibermacine, R. Fleurquin, and S. Sadou, "A family of languages for architecture constraint specification," *J. Syst. Softw.*, pp. 815–831, 2010.

[7] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse, "Component-based specification of software architecture constraints," in *CBSE*, 2011, pp. 31–40.

[8] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "Reusable architectural decision models for enterprise application development," in *Proceedings of the Quality of software architectures 3rd international conference on Software architectures, components, and applications, Medford, MA USA*. Springer-Verlag, 2007, pp. 15–32.

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

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Chichester, UK: Wiley, 1996.

[11] E. Thomas, *SOA Design Patterns*. Prentice Hall, 2009.

[12] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: Views and beyond," in *Proceedings of the 25th International Conference on Software Engineering (ICSE), May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 740–741.

[13] R. Ramos, O. Barais, and J. marc Jzquel, "Matching model snippets," in *In: MoDELS07: 10th Int. Conf. on Model Driven Engineering Languages and Systems, Nashville USA*, 2007, p. 15.

[14] R. B. France, D. kyoo Kim, S. Ghosh, and E. Song, "A uml-based pattern specification technique," *IEEE Transactions on Software Engineering*, pp. 193–206, 2004.

[15] M. Beisiegel, D. Booz, S. TIBCO, M. BEA, C. Sharp, and . SAP, "SCA service component architecture," *Assembly Model Specification*, 2007.

[16] OMG, "Object Constraint Language, OCL Version 2.0, formal/2006-05-01," http://www.omg.org/spec/OCL/2.0/, OMG, Tech. Rep., 2006.

[17] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: a qvt-like transformation language," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. ACM, 2006, pp. 719–720.

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

[19] P. Kruchten, P. Lago, and H. van Vliet, "Building up and reasoning about architectural knowledge," *Quality of Software Architectures*, pp. 43–58, 2006.

[20] A. Zalewski, S. Kijas, and D. Sokolowska, "Capturing architecture evolution with maps of architectural decisions 2.0," in *Proceedings of the 5th European conference on Software architecture*, ser. ECSA'11. Springer-Verlag, 2011, pp. 83–96.

[21] R. Capilla, F. Nava, S. Prez, and J. Dueas, "A web-based tool for managing architectural design decisions," *ACM SIGSOFT software engineering notes*, p. 4, 2006.

[22] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software*, pp. 918–934, 2007.

[23] 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 *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, ser. WICSA '08. IEEE Computer Society, 2008, pp. 157–166.

[24] N. B. Harrison and P. Avgeriou, "Leveraging architecture patterns to satisfy quality attributes," in *ECSA*, 2007, pp. 263–270.

[25] M. Elaasar, L. C. Briand, and Y. Labiche, "A metamodeling approach to pattern specification," in *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*. Springer-Verlag, 2006, pp. 484–498.