Modeling Variants of Architectural Patterns

Waqas Kamal, Ahmad; Avgeriou, Paris; Zdun, Uwe

Published in:
EPRINTS-BOOK-TITLE

Modeling Variants of Architectural Patterns

Ahmad Waqas Kamal[1], Paris Avgeriou[1], Uwe Zdun[2]

[1]Department of Mathematics and Computer Science
University of Groningen, the Netherlands
a.w.kamal@rug.nl, paris@cs.rug.nl

[2]Information Systems Institute
Vienna University of Technology
Vienna, Austria
zdun@infosys.tuwien.ac.at

Abstract. Systematic modeling of architectural patterns is a challenging task mostly because of the inherent pattern variability and because pattern elements do not match the architectural abstractions of modeling languages. In this paper, we describe an approach for systematic modeling of architectural patterns using a set of architectural primitives and a vocabulary of pattern-specific components and connectors. These architectural primitives can be used as the basic building blocks for modeling a number of architectural patterns. We introduce a profile for the UML2 meta-model to express the architectural primitives. The use of the primitives along with the stereotyping scheme is capable of handling some of the challenges for the systematic modeling of architectural patterns, such as expressing pattern participants in software architecture.

Keywords: Architectural Pattern, Architectural Primitive, Modeling, UML.

1 Introduction

Architectural patterns provide solutions to recurring problems at the architecture design level. Such patterns help architects to design software architectures by providing proven solutions to recurring design problems. So far, a huge list of patterns has been documented in the literature [10, 16]. These patterns have been successfully applied to design software in different domains and provide concrete guidelines for modeling structural and behavioral aspects of software systems. Although at present, the practice of modeling architectural patterns is largely unsystematic and ad hoc, the topic of systematic pattern modeling is receiving increasing attention from researchers and practitioners [9].

In spite of the benefits that patterns offer for solving recurring problems and the ever-growing list of documented patterns, there is not yet a proven approach for the systematic modeling of architectural patterns and pattern variants in software design. Some architecture description languages (ADLs), such as UniCon [15], Aesop [12], ACME [13], and Wright [14] capture specific concepts for modeling patterns. However, none of the approaches presented so far for modeling architectural patterns can fully express architectural patterns [3]. This is because each pattern addresses a whole solution space comprised of different solution variants of the same pattern, which are difficult to express in a specific ADL. In contrast to ADLs, UML offers a generalized set of elements to describe software architecture but UML's support for modeling patterns is weak because the pattern elements do not match the architectural abstractions provided in UML. In summary, both ADLs and UML provide only limited support for modeling patterns.

In our previous work [2], we have identified a set of *architectural primitives*. These primitives offer reusable modeling abstractions that can be used to systematically model the

solutions that are repetitively found in different patterns. In this paper, we introduce a few more primitives and use all the primitives discovered during our current and previous work to devise an approach that is capable of systematically modeling architectural patterns in system design. The main contribution of this paper lies in modeling pattern variability using the primitives, identifying the pattern aspects that are difficult to express using primitives, and devising a generalized scheme that uses a vocabulary of pattern-elements specific components and connectors (e.g., pipes, filters) and primitives in conjunction for systematically modeling architectural patterns.

The remainder of this paper is structured as follows: in Section 2 we present our approach for representing patterns and primitives as modeling abstractions, exemplified using an extension of the UML. Section 3 gives detailed information of the primitives discovered during our work and briefly introduces the primitives discovered in our previous work. Section 4 gives an overview of the relationships between patterns and primitives. Section 5 describes the modeling of a few selected pattern variants using primitives and a design elements vocabulary. Section 6 mentions the related work and Section 7 discusses future work and concludes this study.

2 Extending UML to Represent Patterns and Primitives

UML is widely known as an industry standard modeling language and is highly extensible [17]. There are two approaches for extending UML: extending the core UML metamodel or creating profiles which extend metaclasses. Our work focuses on the second approach where we create profiles specific to the individual architectural primitives. Although this work is exemplified using the UML 2.0, the same approach can be used for other modeling languages as long as the selected modeling language supports an extension mechanism to handle the semantics of the primitives. The key idea is that the modeling language can be extended to facilitate the semantics of the architectural primitives and that these primitives can then be used to model patterns.

We extend the UML metamodel for each discovered architectural primitive using a UML profile. That is, we define the primitive as extensions of existing metaclasses of the UML using stereotypes, tagged values, and constraints:

- *Stereotypes:* Stereotypes are one of the extension mechanisms to extend UML metaclasses. We use stereotypes to extend the properties of existing UML metaclasses. For instance, the Connector metaclass is extended to generate a variety of primitive-specific, specialized connectors.

- *Constraints:* We use Object Constraint Language (OCL) [6] to place additional semantic restrictions on extended UML elements. For instance, constraints can be defined on associations between components, navigability, direction of communication, etc.

- *Tagged Values* allow one to associate tags to architectural elements. For example, tags can be defined to represent individual layers in a layered architecture using layer numbers.

We chose the UML profiles extension mechanism due to the following reasons:

- A large community of software architects understands UML as an industry standard modeling language. This gives an opportunity to use the existing set of UML elements as extensions to create stereotypes. Thus, time to learn a new language and the risks of a novel approach are reduced.

- Profiles are good enough to serve for this purpose. UML allows creating profiles without changing the semantics of the underlying elements of the UML metamodel.

- A number of UML tools are available to design software architecture and support profiles out-of-the-box. In contrast, a metamodel extension would require an extension of the tools.

In the architectural primitives, presented in this paper, we mainly extend the following classes of the UML 2 metamodel to express the primitives:

- *Components* are associated with required and provided interfaces and own ports. Components use connectors to connect with other components or with its internal ports.

- *Interfaces* provide contracts that classes (and components as their specialization) must comply with. We use the interface meta-class to support provided and required interfaces, where provided interface represent functions offered by a component and required interface represents functions expected by component from its environment.

- *Ports* are the distinct point of interaction between the component that owns the ports and its environment. Ports specify the required and provided interfaces of the component that owns them.

- *Connectors* connect the required interfaces of one component to the provided interfaces of other matching components.

3 Architectural Primitives

This section provides an extension to our previous work [2] where we have listed nine architectural primitives. We first present seven primitives discovered in the Component-Connector view that are repetitively found as abstractions in modeling variants of a number of patterns. The aim of our work is to capture common recurring solutions at an abstraction level that can be used to model a number of pattern variants, hence providing a better reusability and systematic support to model abstractions for patterns.

3.1. Previous Work

In our previous work, we have presented nine primitives mainly discovered in the Component-Connector view [2]. In the upcoming sections, we use all the current and the previously discovered primitives to show the systematic modeling of patterns. To avoid repetition, following we give a brief description of each primitive discovered during our previous work.

- *Callback:* A component B invokes an operation on Component A, where Component B keeps a reference to component A – in order to call back to component A later in time.

- *Indirection:* A component receiving invocations does not handle the invocations on its own, but instead redirects them to another target component.

- *Grouping:* Grouping represents a Whole-Part structure where one or more components work as a Whole while other components are its parts.

- *Layering:* Layering extends the Grouping primitive, and the participating components follow certain rules, such as the restriction not to bypass lower layer components.

- *Aggregation Cascade:* A composite component consists of a number of subparts, and there is the constraint that composite A can only aggregate components of type B, B only C, etc.

- *Composition Cascade:* A Composition Cascade extends Aggregation Cascade by the further constraint that a component can only be part of one composite at any time.

- *Shield:* Shield components protect other components from direct access by the external client. The protected components can only be accessed through Shield.

- *Typing:* Using associations custom typing models are defined with the notion of super type connectors and type connectors.

- *Virtual Connector:* Virtual connectors reflect indirect communication links among components for which at least one additional path exists from the source to the target component.

3.2. Description and Modeling Solutions to the Architectural Primitives in the Component-Connector View

In this section, we present six primitives that are repetitively found among a number of architectural patterns. For each discovered primitive, we briefly describe the primitive, discuss the issues of modeling these primitives in UML, present UML profile elements as a concrete modeling solution for expressing the primitive, and motivate known uses of the primitive in architectural patterns.

I. Push-Pull

Context: Push, Pull, and Push-Pull structures are common abstractions in many software patterns. They occur when a target component receives a message on behalf of a source component (Push), or when a receiver receives information by generating a request (Pull). Both structures can also occur together at the same time (Push-Pull).

Modeling Issues: Semantics of push-pull structures are missing in UML diagrams. It is difficult to understand whether a certain operation is used to push data, pull data, or both. A major problem in modeling the patterns using Pushs or Pulls in UML is that although Push-

Pull structures are often used to transmit data among components, it cannot be explicitly modeled in UML.

Modeling Solution: To capture the semantics of Push-Pull properly in UML, we propose a number of new stereotypes for dealing with the three cases Push, Pull, and Push-Pull. Figure 1 illustrates these stereotypes according to the UML 2.0 profile package, while Figures 2 and 3 depict the notation used for the stereotypes.

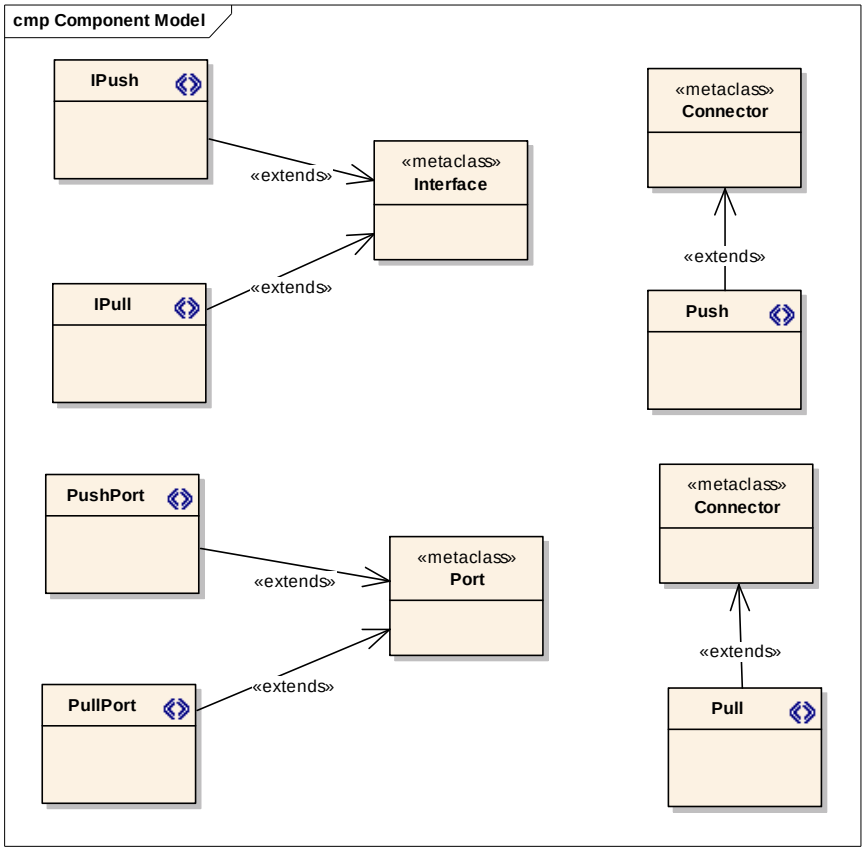

Fig. 1 Stereotypes for modeling Push-Pull

The Push-Pull primitive consists of the following stereotypes and constraints:

- «*IPush*»*:* A stereotype that extends the 'Interface' metaclass and contains methods that Push data among components.

- «*IPull*»*:* A stereotype that extends the 'Interface' metaclass and contains methods that Pull data among components.

- «*PushPort*»*:* A stereotype that extends the 'Port' metaclass and is supported by IPush as *provided* interface and IPull as *required* interface. This can be formalized using two OCL constraints:

```
-- A Push port is typed by IPush as a
-- provided interface

inv: self.basePort.provided->size() = 1
```

```
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IPush.baseInterface->exists (j | j=i)


-- A Push port is typed by IPull as a
-- required interface

inv: self.basePort.required->size() = 1
  and self.basePort.required->forAll(
    i:Core::Interface |
      IPull.baseInterface->exists (j | j=i)
```

- «*PullPort*»*:* A stereotype that extends the port metaclass and is supported by IPush as *required* interface and IPull as *provided* interface. This can be formalized using two OCL constraints for the Pull port:

```
-- A Pull port is typed by IPull as a
-- provided interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IPull.baseInterface->exists (j | j=i)


-- A Pull port is typed by IPush as a
-- required interface

inv: self.basePort.required->size() = 1
  and self.basePort.required->forAll(
    i:Core::Interface |
      IPush.baseInterface->exists (j | j=i)
```

- «*Push*»*:* A stereotype that extends the ‘Connector’ metaclass and connects a PushPort with a matching PullPort of another component.

```
-- A Push connector has only two ends.

inv: self.baseConnector.end->size() = 2

-- A Push connector connects a PushPort
-- of a component to a matching PullPort of
-- another component. A PushPort matches a
-- PullPort if the provided interface of the former
-- matches the required interface of the later

inv: self.baseConnector.end->forAll(
  e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
    (e1.role->notEmpty() and
     e2.role->notEmpty()) and
    (if PushPort.basePort->exists(p |
      p.oclAsType(Core::ConnectableElement) =
      e1.role)
    then
      (PullPort.basePort->exists(p |
```

```
      p.oclAsType(Core::ConnectableElement) =
      e2.role)
    and
    e1.role.oclAsType(Core::Port).required =
    e2.role.oclAsType(Core::Port).provided)
  else
    PullPort.basePort->exists(p|
      p.oclAsType(Core::ConnectableElement) =
      e1.role)
  endif)))
```

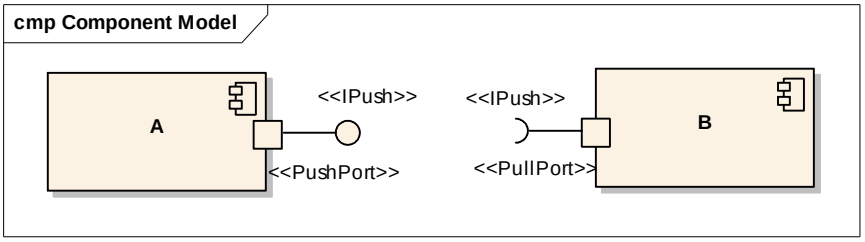

Fig. 2 Ports and interfaces to model Push structure from B to A

- «*Pull*»*:* A stereotype that extends the 'Connector' metaclass and connects a PullPort with a matching PushPort of another component.

```
-- A Pull connector has only two ends.
inv: self.baseConnector.end->size() = 2


-- A Pull connector connects a PullPort
-- of a component to a matching PushPort of
-- another component. A PushPort matches a
-- PullPort if the provided interface of the former
-- matches the required interface of the later

inv: self.baseConnector.end->forAll(
  e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
    (e1.role->notEmpty() and
     e2.role->notEmpty() ) and
    (if PushPort.basePort->exists(p |
      p.oclAsType(Core::ConnectableElement) =
      e1.role)
    then
      (PullPort.basePort->exists(p |
        p.oclAsType(Core::ConnectableElement) =
        e2.role)
      and
      e1.role.oclAsType(Core::Port).required =
      e2.role.oclAsType(Core::Port).provided)
    else
      PullPort.basePort->exists(p|
        p.oclAsType(Core::ConnectableElement) =
        e1.role)
    endif)))
```

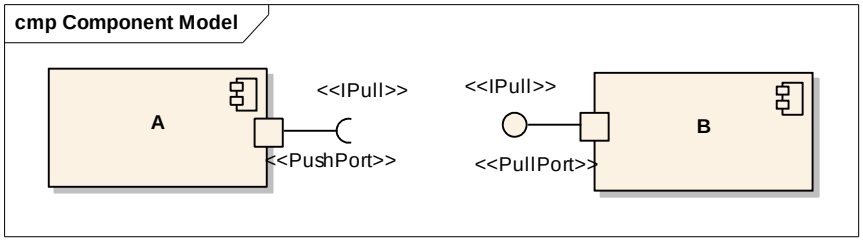

Fig. 3 Ports and interfaces to model Pull structure from A to B

Known uses in patterns:

- In the Model-View-Controller [11] pattern, the model pushes data to the view, and the view can pull data from the model.
- In the Pipes and Filters [11] pattern, filters push data, which is transmitted by pipes to other filters. In addition, pipes can request data from source filters (Pull) to transmit it to the target filters.
- In the Publish-Subscribe [11] pattern, data is pushed from a framework to subscribers and subscribers can pull data from the framework.
- In the Client-Server [11] pattern, data is pushed from the server to the client, and the client can send a request to pull data from the server.

II. Virtual Callback

Context: Consider two components are connected via a callback mechanism. In many cases the callback between components does not exist directly, rather there exist mediator components between the source and the target components. Such information is should be represented at the design level. For instance, in the MVC pattern, a model may call a view to update its data but this data may be rendered first by the mediator components before it is displayed on the GUI.

Modeling Issues: The virtual relationship is an important aspect to show collaborating elements. The standard UML supports connector or association links to model virtual relationships. However, such a relationship cannot be made explicit in standard UML as it may become difficult to determine which components have subscribed to other components to be called back virtually.

Modeling Solution: To capture the semantics of Virtual Callback properly in UML, we extend the Callback [2] primitive using following constraints:

```
-- A VirtualCallback can only be used between
-- two components A and B, if there is a path of
-- components and connectors that link A to B

inv: self.baseConnector.end.role.oclAsType(
  Core::Property).class->forAll(
    c1,c2:Core::Component | c1 <> c2 implies
      c1.oclAsType(Core::Component).connects(c2))
```

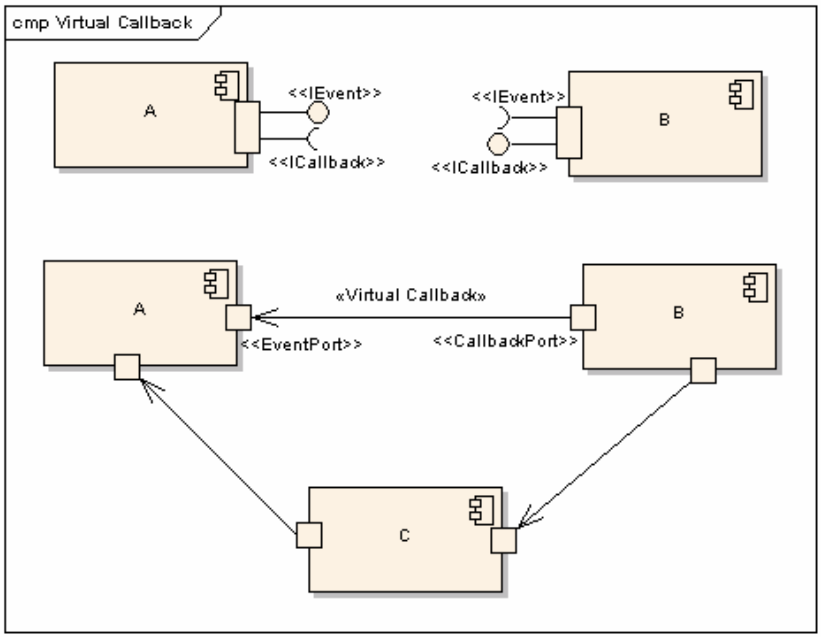

Figure 4. The notation of the stereotypes in Virtual Callback Modeling

Known Uses in Patterns:

- In the MVC [11] pattern, the view and model components may communicate to each other virtually using callback operation.
- In the Observer [11] pattern, the subjects may observe the target objects virtually.
- In the Publish-Subscribe [11] pattern, the publishers may callback subscribers virtually.

III. Adaptor

Context: This primitive converts the provided interface of a component into the interface the clients expect. The adaptor primitive is a close match to the Object Adaptor [16] pattern.

Modeling Issues: Adaptors shield the underlying system implementation from its surroundings. However, adaptors can not be explicitly modeled using the architectural abstractions present in UML as their task is more focused on conversion rather than computation.

Modeling Solution: To capture the semantics of Adaptor properly in UML, we propose following new stereotypes:

- «IAdaptor» extends the interface metaclass and types AdaptorPort with the provided interface and AdapteePort with the required interface.

- «IAdaptee» extends the interface metaclass and types AdapteePort with the provided interface and AdaptorPort with the required interface.

- «AdaptorPort» extends the Port metaclass and is typed by the IAdaptor provided interface and IAdaptee required interface. This can be formalized using following OCL constraints:

```
-- AdaptorPort is typed by IAdaptor as a
-- provided interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IAdaptor.baseInterface->exists (j | j=i)

-- AdaptorPort is typed by IAdaptee as a
-- required interface

inv: self.basePort.required->size() = 1
  and self.basePort.required->forAll(
    i:Core::Interface |
      IAdaptee.baseInterface->exists (j | j=i)
```

- «AdapteePort» extends the Port metaclass and is typed by the IAdaptor required interface and IAdaptee provided interface. This can be formalized using following OCL constraints:

```
-- AdapteePort is typed by IAdaptee as a
-- provided interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IAdaptee.baseInterface->exists (j | j=i)

-- AdapteePort is typed by IAdaptor as a
-- required interface

inv: self.basePort.required->size() = 1
  and self.basePort.required->forAll(
    i:Core::Interface |
      IAdaptor.baseInterface->exists (j | j=i)
```

- «Adaptor» extends the component metaclass and attaches AdaptorPort.

```
inv: self.baseComponent.ownedPort.name = 'AdaptorPort'
```

Known Uses in Patterns:

- In the Layers [11] pattern, the adaptor supports the separation of explicit interface of a layer from its implementation [16].
- In the Broker [11] pattern, the adaptor translates the messages coming from remote services to the underlying system.
- In the Microkernel [11] pattern, the adaptor is used to map communication between external and internal servers.
- In the Proxy [11] pattern, the adaptor is used to separate the interface from the implementation.

IV. Passive Element

Context: Consider an element is invoked by other elements to perform certain operations. Passive elements do not call operations of other elements.

Modeling Issues: UML components do not structurally differentiate between active and passive elements. Such a differentiation is important to understand clearly the responsibility of individual elements in the design.

Modeling Solution: To capture the semantics of Passive Element properly in UML, we propose the following new stereotypes:

- «IPassive» extends the Interface metaclass and types the PassivePort.

- «PassivePort» extends the Port metaclass and supports the IPassive provided interface. This can be formalized with OCL as follows:

```
-- A PassivePort provides IPassive interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IPush.baseInterface->exists (j | j=i)
```

- «PElement» extends the Component metaclass and attaches the PassivePort

```
inv: self.baseComponent.ownedPort.name = 'PassivePort'
```

Known Uses in Patterns:

- In the Pipe-Filter [11] pattern, the passive filter cannot pull or push data to its neighboring filters.
- In the MVC [11] pattern, the passive view only receives or displays data to the user and does not invoke any operation on a model or controller elements.

V. Control

Context: Calling a method in the target component may involve transfer of control from the source to the target component.

Modeling Issues: A control typically differs from data, events, and other forms of communications among components. However, UML elements cannot structurally express the presence of control operations in software design.

Modeling Solution: To capture the semantics of Control primitive properly in UML, we propose following new stereotypes:

- «IControl» extends the interface metaclass and types TPort with the provided interface and SPort with the required interface.

- «TPort» extends the Port metaclass and is typed by the IPControl provided interface and IRControl required interface.

```
-- A Target port is typed by IPControl as a
-- provided interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
```

```
    i:Core::Interface |
      IPControl.baseInterface->exists (j | j=i)
```

- «SPort» extends the Port metaclass and is typed by the IRControl required interface.

```
-- A Source port is typed by IRControl as a
-- required interface

inv: self.basePort.required->size() = 1
  and self.basePort.required->forAll(
    i:Core::Interface |
      IRControl.baseInterface->exists (j | j=i)
```

- «Control» extends the connector metaclass and provides a connection from the source port (SPort) to the target port (TPort).

```
-- A Control connector has only two ends
inv: self.baseConnector.end->size() = 2

-- A Control connector connects the Source port of a
-- component to the Target port of another component

inv: self.baseConnector.end->forAll(
  e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
    (e1.role->notEmpty() and
      e2.role->notEmpty() ) and
    (if SPort.basePort->exists(p |
      p.oclAsType(Core::ConnectableElement) =
      e1.role)
    then
      (TPort.basePort->exists(p |
        p.oclAsType(Core::ConnectableElement) =
        e2.role)
      and
      e1.role.oclAsType(Core::Port).required =
      e2.role.oclAsType(Core::Port).provided)
    else
      TPort.basePort->exists(p|
        p.oclAsType(Core::ConnectableElement) =
        e1.role)
    endif)))
```

Known Uses in Patterns:

- In the MVC [11] pattern, controller translates user events into actions that are performed on model.
- In the PAC [11] pattern, a controller in one PAC agent is connected to other PAC agents in the hierarchy using control link.
- In the Layers [11] pattern, an upper layer invokes operation on the sub-sequent underneath layer. The operation is usually performed through a control to invoke specific operations.

VI. Mediator

Context: Sometimes certain objects in the set of objects cooperate with several other objects. Allowing direct link between such objects can overly complicate the communication and result in strong coupling between objects [16]. To solve this problem, mediator components are used.

Modeling Issues: Mediator objects are typically involved in decoupling objects and store the collective behavior of interacting components. The structural representation of mediator components in UML diagrams is hard to understand.

Modeling Solution: To capture the semantics of Mediator primitive properly in UML, we propose following new stereotypes:

- «IRMediator» extends the interface metaclass and types the port MRPort.

- «IFMediator» extends the interface metaclass and types the port MFPort.

- «MRPort» extends the Port metaclass and is typed by the provided interface IRMediator.

```
-- A MRPort is typed by IRMediator as a
-- provided interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IRMedator.baseInterface->exists (j | j=i)
```

- «MFPort» extends the Port metaclass and is typed by the provided interface IFMediator

```
-- A MFPort is typed by IFMediator as a
-- provided interface

inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      IFMediator.baseInterface->exists (j | j=i)
```

- «Mediator» extends the Component metaclass and attaches MFPort and MFPort

```
-- A Mediator component owns ports to
-- connect source and target components

inv: self.baseComponent.ownedPort.name = 'MFPort'
  and self.baseComponent.ownedPort.name = 'MRPort'
```

Known Uses in Patterns:

- In the PAC [11] pattern, a controller is used to mediate communication between agents in the PAC hierarchy.
- In the Microkernel [11] pattern, a mediator receives requests from external server and dispatches these requests to one or more internal servers.

4. The Pattern-Primitive Relationship

Architectural patterns and architectural primitives are complementary concepts. Modeling patterns in a system design is applying one of the alternate solutions to solve specific problems at hand [11]. While, primitives serve as the building blocks for expressing architectural patterns. In this context, patterns offer general solutions while primitives offer relatively more specific solutions. Similar to the selection of architectural patterns among complementary patterns, primitives might also need to be selected among complementary primitives, e.g., based on the system requirements you might choose either Shield or Indirection. Such a decision to select appropriate primitive involves the context in which the pattern is applied, and the specific solution variant addressed by the pattern. Moreover, certain primitives can be used in combination with other primitives. For example, the Callback and Push-Pull primitives can work in conjunction to serve a common purpose.

Table 1 provides a patterns-to-primitives-mapping, which is based on the primitives discovered so far in our work. The detailed discussion about the discovery of each primitive in the related patterns is already documented in the 'Known Uses in Patterns' subsections of our current and previous work (see Section 3 and [2] for details). The intention is to use the pool of all available primitives to model architectural patterns. However, the mapping from patterns to primitives is not one-to-one: rather different variants of the patterns can be modeled using a different combination of primitives. Thus, the decision to apply a specific primitive for modeling patterns lies with the architect who selects primitive(s) that best meet the needs to model the selected pattern(s).

The issues addressed above directly deal with the traditional challenge of modeling pattern variability. The solution variants entailed by a pattern can be applied in infinite different ways and so is the selection of primitives for modeling pattern variants. More important is that whichever pattern variant is applied in system design, it should address the solution clearly with structural and semantic presence. Using our primitives allows the architect to apply a near infinite solution variants with certain level of reusability. Such a reusability support also depends on the context in which the pattern is applied as in some cases extra constraints or missing pattern semantics may be required.

Table 1 shows a list of architectural primitives that can be selected to model the associated patterns.

Patterns \ **Primitives**	Callback	Indirection	Grouping	Layering	Aggregation Cascade	Composition Cascade	Shield	Typing	Virtual Connector	Push	Pull	Virtual Callback	Adaptor	Passive Element	Control	Mediator
Observer	X											X				
MVC	X									X	X	X		X	X	
Reactor	X															
Event	X															
Interceptor	X															
Visitor	X															
Layers		X	X	X			X						X		X	

Indirection Layer		X		X											
Virtual Machine		X													
Interpreter		X													
Adaptor		X										X			
Façade		X	X				X								
Proxy		X													
Component Wrapper		X													
Wrapper façade		X													
Broker			X				X		X						
Layered System				X								X			
Object System Layer				X			X	X							
Composite					X	X									
Cascade					X	X									
Organization Hierarchy					X	X									
Message Redirector							X								
Type Object								X							
Knowledge Level								X							
Remoting Patterns							X		X						
Remote Proxy									X						
Client-Server	X		X	X					X	X	X	X			
Publish-Subscribe	X									X	X	X	X		
PAC	X													X	X
Active Repository	X														
Microkernel		X													
Pipe-Filter										X	X			X	
Reflection				X											
Explicit Invocation	X														

Table 1. Patterns to Primitives Mapping

*Expressing Missing Pattern Semantics in UML: A*n important aspect of modeling architectural patterns is the explicit demonstration of patterns in system design and support for automated model validation. Such a representation helps in better understanding of the system by allowing the user to visualize and validate the patterns modeled in a system design. The primitives described above capture recurring building blocks found in different patterns. However, it may be the case that certain pattern aspects of a specific solution variant may not be fully expressed by the existing set of primitives. Therefore, for expressing missing pattern semantics that are not covered by the primitives, we provide support to the user with a vocabulary of design elements that can be used alongside with the primitives to fully express pattern semantics such as pipes, filters, client, server etc. For this purpose, we stereotype a few UML elements with known semantics of the selected architectural patterns. For instance, a component can be stereotyped as filter and a connector can be stereotyped as pipe. The stereotyping scheme presented here is further complimented by using these stereotypes for modeling the example patterns in next section.

The use of design elements for expressing pattern elements has a number of significant benefits. First, it offers reusability support for expressing patterns in system design. The well-known properties entailed by documented pattern variants can be reapplied in system design as a solution to new problems. Second, this makes it easier for a stakeholder to understand design of the system. For example, the use of design vocabulary to express pipes and filters in system design makes the architecture more explicit to understand and the way different design elements fit in the structure. Third, it offers a good support for automated model validation by

ensuring that selected patterns are correctly applied in a system design. All of these three benefits compliment our use of primitives for modeling patterns. The intention is that though primitives offer good reusability and model validation support, as advocated in our current and previous work [2], the stereotyping scheme presented in this section makes the story complete for the systematic modeling of architectural patterns in a system design.

5. Modeling Architectural Patterns Using Primitives

In this section, we use the primitives and stereotyping scheme described in the previous sections to model specific pattern variants. The patterns modeled in this section are specialization to the patterns documented in POSA [11] and hence are called pattern variants. We do not claim to cover all the variability aspects of the selected patterns. However, an effort to describe selected pattern variants using primitives provides a solid base for modeling unknown pattern variants as well. To serve this purpose, we have selected three traditional architectural patterns namely Layers, Pipe-Filter, and Model-View-Controller (MVC). Moreover, we highlight the pros and cons of modeling these patterns using primitives. This is because each architectural pattern exists with many variant solutions, which are challenging to describe using a predefined modeling approach. We use following guidelines to model each selected pattern variant:

- A brief description of the selected pattern variant
- Mapping selected pattern variant to the list of available primitives
- Highlight the problems in modeling the selected pattern variant using primitives
- Use stereotyping scheme to capture the missing pattern semantics.

5.1. Pipe-Filter

The Pipe-Filter pattern consists of a chain of data processing filters, which are connected through pipes. The filters pass the output through pipes to the adjacent filters. The elements in the Pipe-Filter pattern can vary in the functions they perform. For instance, pipes can buffer data, feedback loops, forks, and active or passive filters etc. The primitives discovered so far address many such variations for systematically modeling Pipe-Filter pattern. However, certain aspects of the Pipe-Filter pattern may not be fully expressed by the primitives e.g. feedback loops, forks, etc. The requirements we consider in this section for modeling the specific pipe-filter pattern variant are: a) filters can push or pull data from the adjacent filters; b) filters can behave as active or passive elements; and c) feedback loop.

At first, we map the selected Pipe-Filter pattern variant to the list of available primitives. We select the Push, Pull, and Passive Element primitives from the existing pool of primitives. The rationale behind the selection of these primitives is as follows:

- The Push and Pull primitives are used to express the pipes that transmit streams of data between filters.
- The filters that are not involved in invoking any operations on their surrounding elements are expressed using the Passive Element primitive.

Modeling Problem: As described in section four, the challenge to model missing pattern semantics is solved by stereotyping UML elements. In the current example, the selected primitives are sufficient to express the Push, Pull, and Passive Elements in the Pipe-Filter pattern. However, we identify that the feedback loop cannot be fully expressed using the existing set of primitives. The existing primitives can express that the data is pushed or pulled

between the filters but this does not express the presence of a feedback structure. Similarly, the semantics of the Pipe and Filter elements are not applied using the existing set of primitives.

Modeling Solution: We apply the Feedback stereotype on the Push primitive to capture the structural presence of feedback loop in the Pipe-Filter pattern. Such a structure represents that the data is pushed from one filter to another filter using the feedback loop. The original Push primitive, as described in section four, extends the UML metaclasses of connector, interface, and port. While the feedback stereotype further specializes the Push primitive by labeling it as Feedback. The introduction of feedback stereotype does not introduce new constraints nor affects the underlying semantics of the Push primitive. Figure 5 shows the stereotypes used for expressing Pipe-Filter pattern.

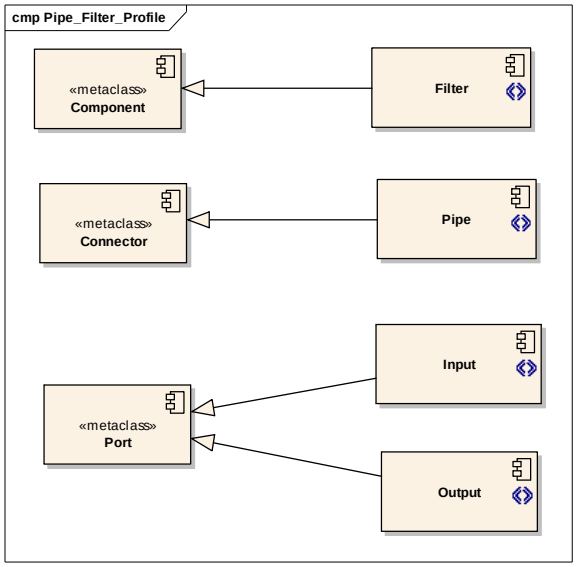

Fig. 5 UML stereotypes for expressing Pipe-Filter pattern participants

«Feedback»: A stereotype that is applied on the Push primitive for expressing the Feedback structure in the Pipe-Filter pattern variant. Feedback stereotype extends the Connector metaclass of UML.

The second stereotype named ‘Filter’ that we use from the existing vocabulary of design elements is defined as follows:

«Filter»: A stereotype that extends the Component metaclass of UML and attaches input and output ports.

A Filter component is formalized using following OCL constraints:

```
-- An Input port is typed by Iinput as a provided
-- interface
inv: self.basePort.provided->size() = 1
  and self.basePort.provided->forAll(
     i:Core::Interface |
        Iinput.baseInterface->exists(j | j = i)

-- An Output port is typed by Ioutput as a required
-- interface
```

```
inv: self.basePort.required->size() = 1
  and self.basePort.provided->forAll(
    i:Core::Interface |
      Ioutput.baseInterface->exists(j | j = i)
```

The third stereotype that we use from the existing vocabulary of design elements is ‘Pipe’ that is defined as follows:

«Pipe»: A stereotype that extends the Connector metaclass of UML and connects the output port of one component to the input port of another component.

A Pipe is formalized using following OCL constraints:

```
-- A Pipe has only two ends.
inv: self.baseConnector.end->size() = 2
```

As shown in the figure below, the first filter in the chain works as a passive filter and does not invoke any operations on its surrounding filters. While the second filter is an active filter, which pulls data from the passive filter and after processing pushes this data to the next filter in the chain. The third filter in the chain sends data back to the passive filter for further processing, and sends the final processed data to the sink.

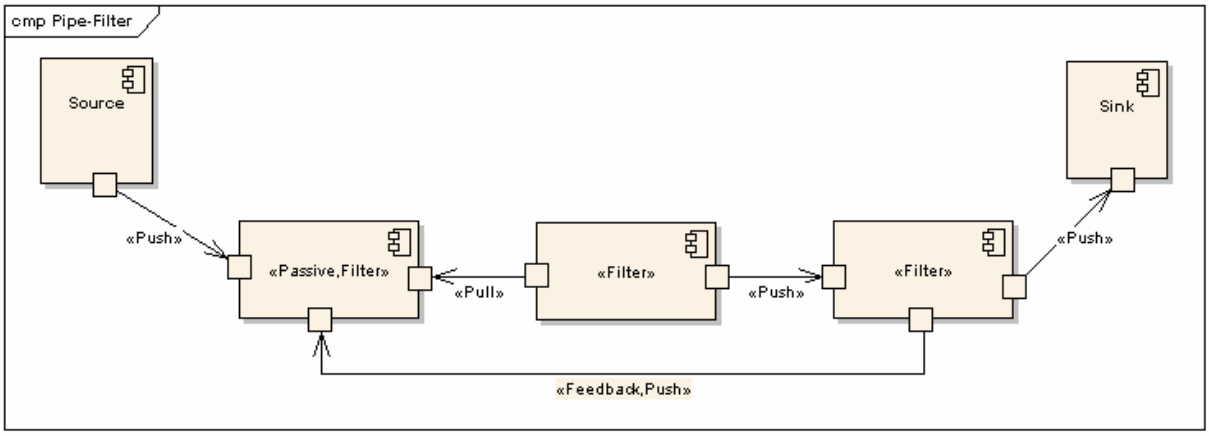

Fig. 6 Modeling Pipe-Filter Pattern Variant Using Primitives

5.2. Model-View-Controller

The structure of MVC pattern consists of three components namely Model, View, and Controller. The Model provides functional core of the application and notifies views about the data change. Views retrieve information from the model and display it to the user. Controllers translate events into requests to perform operations on view and model elements. Usually a change propagation mechanism is used to ensure the consistency between the three components of the MVC pattern [4].

As a first step, we map the MVC pattern to the list of available primitives as shown in the table in section four. We select the callback, passive element and control primitives for modeling the MVC pattern. The rationale behind the selection of these primitives is as follows:

- The view subscribes to the model to be called back when some data change occurs and works as passive object by not invoking any operation on the model.
- Controller issues a command request on the model when some event occurs.

Modeling Problem: However, not every aspect of the MVC pattern can be modeled using the existing set of primitives. For instance, the Model, View, and Controller components are not mapped to any primitives discovered so far. Keeping in view the general nature of these components, there is a need to provide reusability support by making these three pattern elements available in the existing vocabulary of design elements.

Modeling Solution: As described above, despite the reusability support offered by the selected primitives, the MVC pattern semantics are not structurally distinguishable. We use the following three design elements from the existing set of design elements that are stereotyped using UML metaclasses:

«Model»: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with Controller and View components.

«Controller»: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with Model and View components.

«View»: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with Model and Controller components.

As shown in the figure below, the Controller receives input and translates it into requests to the associated model using the Control primitive. While, the Model calls back view when a specific data change occurs.

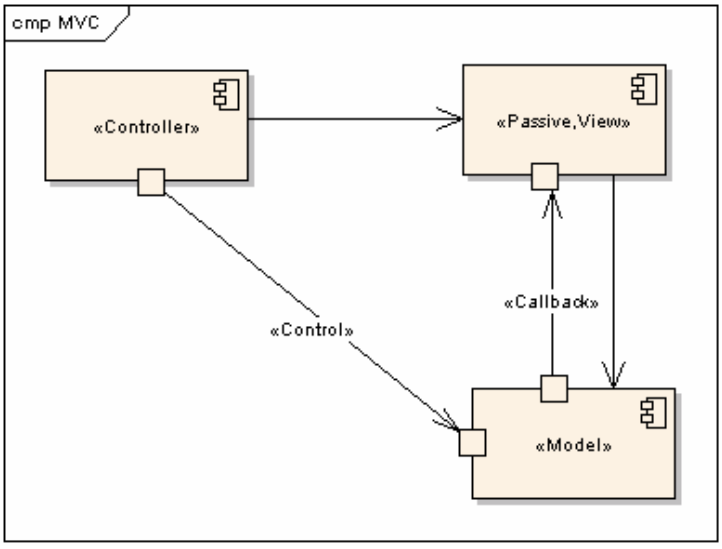

Fig. 7 Modeling MVC Pattern Using Primitives

5.3. Layers

The Layers pattern groups elements at a certain level of abstraction where lower layers provide services to the adjacent upper layer. Such a structure is used to reduce dependencies between objects in different Layers.

As a first step, we map the Layers pattern to the list of available primitives and select the Layering primitive. The rationale behind the selection of this primitive is as follows:

- Components are members of specific layers where each lower layer provides services to the adjacent upper layer
- A component can only be a member of one layer

Modeling Problem: In the Layers pattern, the high-level functions' implementation relies on the lower level ones. Such system requires horizontal partitioning where each partition carries operations at a certain level of abstraction. As each layer in the Layers pattern is a virtual entity so it cannot exist without the presence of at least one component. Moreover, the upper layers cannot bypass the layers for using services in the bottom layers i.e. in Figure 8, the group members from layer3 can call components in layer2, but not into layer1.

Modeling Solution: Modeling the Layers pattern using primitives is a typical example to show the systematic modeling of architectural patterns. Almost all structural characteristics of the Layers pattern are modeled using the Layering primitive as shown in Figure 8. Using the layering primitive, the constraints specification assures that within an individual layer all component work at the same level of abstraction and no component belong to more than one layer at any time. Moreover, no additional stereotyping of UML elements is required to model this specific variant of the layers pattern.

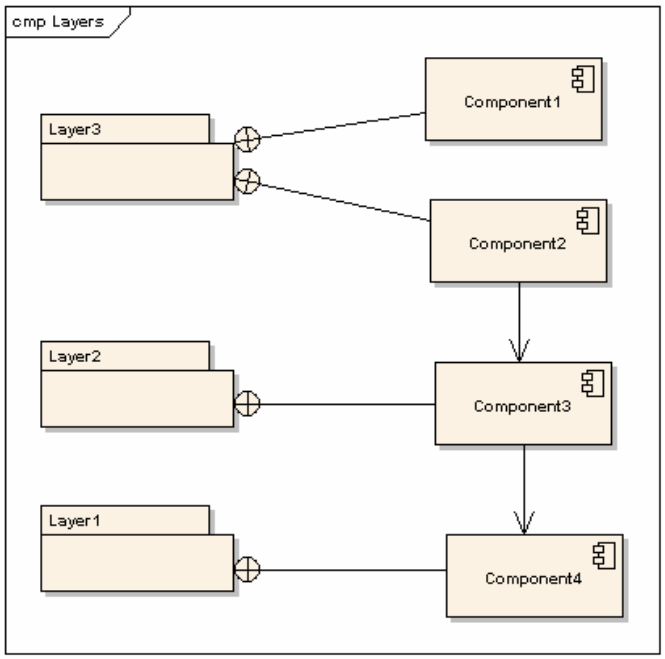

Fig. 8 Modeling Layers Pattern Using Primitives

6. Related Work

The work described in this paper is based on our previous work [2] where we present an initial set of primitives for modeling architectural patterns. However, the idea to use primitives for software design is not novel and has been applied in different software engineering disciplines [5]. The novelty of our work lies in the use of primitives for systematically modeling architectural patterns, which had not be addressed before.

Using different approaches, a few other researchers have been working actively on the systematic modeling of architectural patterns [19, 7]. Garlan et al. [7] proposes an object

model for representing architectural designs. The authors characterize architectural patterns as specialization of the object models. However, each such specialization is built as an independent environment, where each specialization is developed from scratch using basic architectural elements. Our approach significantly differs in a way that our focus is on reusing primitives and pattern elements and only where required we extend the primitives and pattern elements to capture the missing pattern semantics.

Simon et al. [18] extends the UML metamodel by creating pattern-specific profiles. The work by Simon et al. maps the MidArch ADL to the UML metamodel for describing patterns in software design. However, this approach does not address the issue of modeling a variety of patterns documented in the literature rather manual work is required to create profiles for each newly discovered pattern. Our approach distinctively differs from this work as we focus on describing a generalized list of patterns using the primitives.

Mehta et al. [5] propose eight forms and nine functions as basic building blocks to compose pattern elements. Their approach focuses on a small set of primitives for composing elements of architectural styles. Our approach is different in the sense that we offer a more specialized set of primitives that are captured at a rather detail level of abstraction. Moreover, we use vocabulary of pattern elements in parallel to architectural primitives to capture the missing semantics of architectural patterns.

7. Conclusion

Using architectural primitives and a design elements vocabulary in combination offers a systematic way to model patterns in system design. We have extended existing pool of primitives with the discovery of seven more primitives. With the help of few example patterns, we show the realistic approach for modeling architectural patterns using primitives. The scheme to use stereotyping in conjunction with primitives offers: a) reusability support by providing vocabulary of design elements that entail the properties of known pattern participants; b) automated model validation support by ensuring that the patterns are correctly modeled using primitives; and c) explicit representation of architectural patterns in system design.

To express the discovered primitives and design elements vocabulary, we have used UML2.0 for creating profiles. As compared with the earlier versions, UML2.0 has come up with many improvements for expressing architectural elements. However, we still find UML as a weak option in modeling many aspects of architectural patterns e.g. weak connector support. As a solution to this problem, the extension mechanisms of the UML offer an effective way for describing new properties of modeling elements. Moreover, the application of the profiles to the primitives allows us to maintain the integrity of the UML metamodel. By defining primitive-specific profiles, we privilege the user to apply *selective* profiles in the model.

As future work, we would like to advance in the automation of our approach by developing a tool, which supports modeling pattern variability, documentation, analyzing the quality attributes, source code generation, etc. We believe that we can discover more primitives in different architectural views in the near future, which will provide a better re-usability support to the architects for systematically expressing architectural patterns.

References

[1] Paris Avgeriou and Uwe Zdun. Architectural Patterns Revisited - A Pattern Language, In proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLOP), pp. 1-39, Irse, Germany, (2005)

[2] Uwe Zdun and Paris Avgeriou. Modeling Architecture Patterns using Architecture Primitives, OOPSLA' 05, ACM (October 2005)

[3] Ahmad Waqas Kamal, Paris Avgeriou. An evaluation of ADLs on modeling patterns for software architecture design, 4th International Workshop on Rapid Integration of Software Engineering Techniques, Luxembourg (2007)

[4] Neil Harrison and Paris Avgeriou. Pattern-Driven Architectural Partitioning – Balancing Functional and Non-Functional Requirements, First International Workshop on Software Architecture Research and Practice (SARP'07), Silicon Valley, USA, IEEE Computer Society Press. (2007)

[5] N. R. Mehta and N. Medvidovic. Composing Architectural Styles from Architectural Primitives, In Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT international symposium on foundations of software engineering, page. 347-350, Helsinki, Finland, (2005)

[6] Object Constraint Language Specification versions 1.1, OMG standard, http://umlcenter.visual-paradigm.com/umlresources/obje_11.pdf

[7] David Garlan, Robert Allen and John Ockerbloom. Exploiting Style in Architectural Design Environments, In Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering, New Orleans, LA (December 1994)

[8] David Garlan, Robert Allen and John Ockerbloom. Exploiting Style in Architectural Design Environments, In Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering, New Orleans, LA (December 1994)

[9] Morgan Bjorkander and Cris Kobryn. Architecting Systems with UML 2.0, IEEE Computer Society, 0740-7459/03, IEEE (July 2003)

[10] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, John Wiley & Sons, ISBN 978-0-471-48648-0

[11] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. Pattern Oriented Software Architecture: A System of Patterns, John Wiley & Sons, ISBN 0 471 95869 7

[12] David Garlan, Robert Allen and John Ockerbloom. Exploiting Style in Architectural Design Environments, In Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering, New Orleans, LA (December 1994)

[13] David Garlan, Robert Monroe and David Wile. ACME: An Architecture Description Interchange Language, Proceedings of CASCON 97, Toronto, Ontario, pp. 169-183, (January 1997)

[14] Robert Allen and David Garlan. A Formal Basis For Architectural Connection, ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pp. 213-249, (July 1997)

[15] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, Vol. 21, no. 4, pp. 314-335, (April 1995)

[16] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, John Wiley & Sons, ISBN 978-0-470-05902-9

[17] Xzzcz Morgan Bjorkander and Cris Kobryn: Architecting Systems with UML 2.0, IEEE Computer Society, 0740-7459/03, IEEE (July 2003)

[18] Simon Giesecke, Florian Marwede, Matthias Rohr, and Willhelm Hasselbring. A Style-Based Architecture Modeling Approach For UML2 Component Diagrams, In Proceedings of Software Engineering and Applications, SEA 2007, Cambridge, MA, USA, (2007)