Modeling the Variability of Architectural Patterns

Kamal, Ahmad Waqas; Avgeriou, Paris

Published in:
EPRINTS-BOOK-TITLE

Modeling the Variability of Architectural Patterns

Ahmad Waqas Kamal, Paris Avgeriou
Dep. of Mathematics and Computer Science
University of Groningen, the Netherlands
a.w.kamal@rug.nl, paris@cs.rug.nl

ABSTRACT

Architectural patterns provide proven solutions to recurring design problems that arise in a system context. A major challenge for modeling patterns in a system design is effectively expressing pattern variability. However, modeling pattern variability in a system design remains a challenging task mainly because of the infinite pattern variants addressed by each architectural pattern. This paper is an attempt to solve this problem by categorizing the solution participants of patterns. More precisely, we identify variable participants that lead to specializations within individual pattern variants and participants that appear over and over again in the solution specified by several patterns. With examples and a case study, we demonstrate the successful applicability of this approach for designing systems. Using the UML extension mechanism, we offer extensible architectural modeling constructs that can be used for modeling several pattern variants.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Patterns, Languages, Modeling

Keywords

Architectural Patterns, Variability, UML

1. INTRODUCTION

Architectural patterns document successful experiences of designers and provide proven solutions to recurring design problems that arise in a system context. They specify guidelines for designing the structural and behavioral aspects of a system. An architectural pattern details a fundamental solution to a design problem in the form of pre-defined pattern participants (also known as pattern elements [1]) like pattern-specific components, classes, or objects that work together to resolve the identified problem.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

There are four approaches that have been used so far to express the solution specified by a pattern in system design:

- Architectural Description Languages (ADLs) [2] that have been traditionally used for describing software architecture.
- Unified Modeling Language (UML) [3] [4], which is a widely used generic modeling language for designing systems in different domains but is mostly used to design software.
- Formal approaches [5] that specify precise pattern solution to specific problems e.g. pattern-specific components and connectors.
- Informal box and line diagrams that provide little information about actual computation represented by boxes, and the nature of the interactions between them [6].

In spite of the benefits that these 4 approaches offer, there is not yet an established approach for effectively expressing and applying pattern variants in a system design for the following reasons:

- Modeling pattern **variability** effectively in a system design using existing approaches remains a challenging task. A pattern determines only a basic solution to a design problem rather than tangible architectural elements that can be used 'as is' in a design. The solution specified by a pattern provides only the guidelines to solve a design problem and leaves blank spaces that need to be filled in by a software designer [1]. This requires specialization of a pattern's solution in the context of a system design that leads to different variants of a pattern. Current modeling languages provide limited support for expressing pattern variability. UML offers generic architectural elements (i.e. components, connecters) while few other modeling languages (e.g. ACME, Wright, Aesop) provide support for modeling a limited set of architectural patterns but do not address the challenges of modeling different variants of patterns. Similarly, pattern formalization approaches result in formalized solutions of patterns which may narrow the applicability of a pattern in expressing different system-specific specialized solutions. Current approaches are either too generic or provide support for specific design solutions and fall short in offering a mechanism to grasp the whole solution space covered by architectural patterns.

- The model checking support offered by current system design approaches do not focus explicitly on validating patterns in a system design. Some system design approaches provide generic support for constraint checking e.g. CSP [6] for Wright and Object Constraint Language (OCL) [7] for UML. However, the existing approaches are aimed at describing constraints on a system design in general and require extensive effort for creating pattern-specific constraints in a system design.

The challenges described above are already a focus of ongoing work in the field of software design to effectively model patterns within a system design [8] [9]. In this paper, we address these problems with a specific focus on tackling the issue of modeling pattern variability. We propose to address the problem by identifying and categorizing the pattern participants. Specifically, we identify three kinds of pattern participants: a) *generic pattern participants*: a set of pattern-specific elements within the original solution of a pattern that are required to express a pattern in a system design; b) *specialized pattern participants*: architectural elements within pattern variants that specialize generic pattern participants; and c) *architectural primitives*: recurring architectural abstractions found in the solution space of several patterns which are used as key participants in modeling a variety of patterns. The idea is that the use of architectural primitives in combination with the specialized pattern participants provides a valuable approach to effectively express the variants of several patterns.

To describe these concepts in a modeling language, we pick UML which is a widely known language for software design and provides explicit extensibility mechanism to generate modeling constructs. UML includes extension mechanisms like profiles, tagged values and stereotypes that are used to describe the primitives, generic pattern participants and specialize pattern participants in this work. We extend the UML constructs in the Component-Connector view and use the OCL to formalize the interaction among them.

The remainder of this paper is structured as follows: in Section 2 we present our approach for expressing the pattern variants using primitives, generic and specialized pattern participants. In section 3, we detail the extension mechanism of UML and list the UML metaclasses used in this work. Section 4 explains the UML profiles created for expressing the pattern variants and presents modeling of an example pattern variant. In section 5, we provide validation of this work by designing part of large system using pattern variants. Section 6 mentions the related work and Section 7 discusses future work and concludes this study.

2. THE PROPOSED APPROACH

Despite a large list of architectural patterns documented in the literature, patterns are rarely applied in a system design in their original form as the 'forces' that describe the problem addressed by a pattern largely influence its solution. Resolving these 'forces' in the solution space of a pattern yields different pattern variants. For instance, the Tee and Join variant of the Pipes and Filters [1] pattern specializes the Pipe participant while another variant of the same pattern may require the presence of feedback pipes and so on. To serve the purpose of effectively expressing a variety of pattern variants in a system design, we propose to categorize the solution participants of architectural patterns into three groups as follows:

- *Generic Pattern Participants as Variation Points:* The term 'generic pattern participants' refers to the solution participants of a pattern in their original form as documented in the literature e.g. the Filter and Pipe are the solution participants within the Pipes and Filters pattern. We identify the following four categories that we use to define variability on Generic pattern participants:
  - Element Type: Element type refers to variability in the use of pattern participants in context of different system designs e.g. active agents are a specific type [1] within the Active Object Presentation-Abstraction-Control pattern variant and feedback pipes are specific types within the Pipes and Filters pattern variant.
  - Communication: Communication refers to data transfer connections among pattern participants. Variability in communication refers to different kinds of interaction between the interacting elements e.g. the Client an Server participants can interact either Asynchronously or Synchronously within a system design.
  - Data: Variability in data denotes the occurrence of variation on a set of data types used by communication links or computational components. For instance, the Model participant within the MVC pattern can update data on views using Array, String, etc.
  - Interface: Interface is a contract for a component with which surrounding components interact. Interface variability denotes the variation in services offered and required by a component e.g. the Controller participant within the MVC pattern offers event based services to notify the View and Model participants.

  The categories described above help make it explicit where variability is within the solution specified by an architectural pattern. In this work, the locations within the architecture of patterns where variability is found are called variation points. The variation points offer choices in applying patterns to a system design. For instance, the Pipe participant within the Pipes and Filters pattern can have several types like Fork, Join, Feedback etc. Similarly the Model, View, and Controller are marked as variation points within the MVC pattern that can have several different forms depending on the system context.

- *Specialized Pattern Participants within Pattern Variants:* The specialized pattern participants detail the solution structure of pattern variants and relationships to solve specific design problems e.g. the Direct Communication Broker and Adapter Broker pattern variants are specializations to the Broker pattern [1]. Pattern variants can be modeled using one or more specialized pattern participants. This allows a designer to model unique pattern variants by using different combinations of specialized pattern participants.

- *Architectural Primitives:* The primitives are recurring pattern participants discovered within the solution of several patterns. Architectural primitives act as modeling units to express parts of the solution specified by a variety of patterns. Where specialized pattern participants are aimed at defining different variants of a specific pattern, architectural primitives capture parts of the solution specified by several patterns. In our previous work, we have mined and documented several architectural primitives that we use in this work to express pattern variants (see Appendix A for a complete list of primitives). For instance, we have found the Push-Pull primitive as a solution participant to express data transfer among the participants of Pipes and Filters [1], Publish-Subscribe [1], and Client-Server [1] patterns.

Also, in our previous work [10], we have provided a list of the most commonly used architectural patterns by surveying several existing system designs. We define the generic pattern participants of all such well-know patterns which are then specialized to define several other variants of these patterns. However, due to space restriction, in this paper we express selected patterns only. We first work with the MVC pattern and later demonstrate the applicability of this approach by designing part of a large system.

3. UML EXTENSIONS FOR COMPONENT DIAGRAMS

UML is a widely known extensible modeling language [4]. 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 individual architectural patterns that entail stereotypes and constraints to express several variants of the same pattern. In order to facilitate the unambiguous and error-free modeling, we define the semantics of the selected patterns more precisely with the help of OCL.

We extend the UML metamodel for each selected architectural pattern using UML profiles. That is, we define the pattern participants 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 pattern-specific connector types.

- *Constraints:* We use the Object Constraint Language (OCL) [7] to place additional semantic restrictions on extended UML elements. For instance, constraints can be defined on associations between stereotypes, navigability, direction of communication, etc.

- *Tags:* Tagged Values allow one to associate tags to architectural elements. For example, tags can be defined to represent individual layers in a layered architecture, mark the presence of variation points etc.

For the architectural patterns variants, presented in this paper, we mainly extend the following classes of the UML 2 metamodel:

- *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 represents functions offered by a component and required interface represents functions expected by a 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.

3.1 UML Profiles for Defining Architectural Patterns

In this section, we use the UML to describe the approach presented in the previous section for modeling pattern variants in UML's Component-Connector view. Figure 1 shows the general relationships among these concepts in UML. The UML's extension mechanism of stereotypes, constraints, and tagged values is used to express these notions. Defining architectural primitives using UML is already covered in our previous work [8] [11] while in this section we focus on defining the mechanism to express generic and specialized pattern participants. We extend the UML meta-classes in the

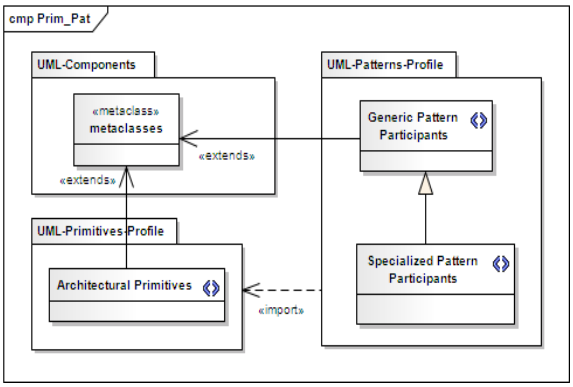

Figure 1: The Relationship between Primitives, Generic and Specialized Participants in UML

Component-Connector view to express generic and specialized pattern participants while the tag values are used to mark the variation points and variants. The pattern participants marked as variation points represent the variation that an element entails for its use in several variants of the same pattern. For instance, in the Model-View-Controller pattern, the Model component marked as variation point is specialized as Document to express the Document-View pattern variant as explained in next section. To serve this purpose, the meta-classes Component, Connector, Port, and Interface are used as described below:

- *Expressing Generic Pattern Participants as Variation Points in UML:* We extend the above mentioned UML meta-classes to express the generic pattern participants as stereotypes. The UML's profile mechanism is used to serve the purpose. For instance, the Pipe and Filter participants of the Pipes and Filters pattern are expressed as stereotypes by extending the Connector and Component meta-classes respectively. To mark selected pattern participants as variation points, the tagged values are defined for pattern participants as a string variable. The variation in UML elements is designated by small dot symbols in UML diagrams used in this work. Although this symbol is not included in UML standard, it has been widely used in literature to denote variation points [12]. We use UML tag syntax *vp <VariationCategory> <Variation1, Variation2, ... VariationN>* to show different variable choices in applying a pattern to a system design.

- *Expressing Specialized Pattern Participants in UML:* We use the UML's inheritance relationship to instantiate several variants from the generic pattern variant participants. For instance, the Feedback and Fork are the specialized pattern participants within the Pipes and Filters pattern that are expressed using the UML's inheritance relationship. Pattern participants that often work in conjunction to model a pattern in system design are expressed using UML's dependency relationship. For example, the Model participant within the MVC pattern has a dependency relationship with data or event ports to communicate with surrounding elements. Furthermore, the same tagged values defined for expressing variation points are overridden to mark the specialized pattern participants.

We also use the following UML metaclasses in order to express the OCL constraints while traversing the UML metamodel: AggregationKind, Classifier, ConnectableElement.

4. DEFINING ARCHITECTURAL PATTERNS IN UML

In this section, we exemplify the approach presented in section 2 and define a well-know architectural pattern namely the Model-View-Controller(MVC) which is one of the most commonly used pattern for designing interactive applications [10] while in the next section, we model part of the design of a large system using pattern variants.

4.1 Defining the MVC pattern in UML

The structure of the MVC pattern consists of three components namely the Model, View, and Controller [1]. The Model provides functional core of the application and updates 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. In such a structure, the Model component provides services to the View and Controller components. Following we use the approach presented in section2 to define participants of the MVC pattern and its variants in UML as shown in Figure 2.

In the solution specified by MVC pattern, the View subscribes to the Model to be called back when some data change occurs. Such a structure can be effectively expressed using the *Callback* primitive. Also, the Controller sends events to the Model for an action to take place, which can be expressed using the *Control* primitive.

The *Callback* and *Control* primitives express part of the solution specified by the MVC pattern. The Model, View and Controller are the generic pattern participants within the MVC pattern that lead to several different forms within individual pattern variants hence marked as variation points. The participants of the MVC pattern use both the event and data based services realized using the ports attached to the Model, View, and Controller components which are used to send/receive data or events. In the specific case of the MVC pattern, the variability in communication is covered by the *Callback* and *Control* primitives and hence not marked in the MVC profile shown in Figure 2.

GenericModel: A stereotype that extends the Component meta-class of UML and attaches ports for interaction with the Controller and View components that is formalized using the following OCL constraints:

```
  Component.allInstances()->iterate(
 i;pairs :  Set(Tuple(c1 :  Component,
  s :  Bag(Component))) = Set |
   let comp :  Component = i.oclAsType(Component),
    stemp :  Bag(Component) =
 comp.ownedConnector->select(j |
  let Callback :  Port = j.oclAsType(
   Connector).end.partWithPort->any(
    owner=i).oclAsType(Port),
     EventPort :  Port =    j.oclAsType(
   Connector).end.partWithPort->any(
    owner<>i).oclAsType(Port) in
if
 j.oclAsType(Connector).getAppliedStereotypes()->
  any( name='Callback ')->notEmpty() and
  EventPort.getOwner() = 'GenericView'and
  Model.ElementType = 'vp'
then
 true
else
 false
endif
```

GenericController: The controller stereotype is an extension to the Component metaclass of UML and attaches ports for interaction with the Model and View components. The controller is formalized using the following OCL constraints:

```
  let comp :  Component = i.oclAsType(Component),
 stemp :  Bag(Component) =
  comp.ownedPort->select(j |
   j.oclAsType(Port), self.ownedPort =
   EventPort and i.oclAsType(Port) in
if
 EventPort.getOwner() = 'GenericView 'and
  Controller.ElementType = 'vp '
then
 true
else
 false
endif
```

GenericView: A stereotype that extends the Component meta-class of UML and attaches ports for interaction with the Controller and Model components which is formalized using the OCL constraints as follows:

```
  let comp :  Component = i.oclAsType(Component),
```

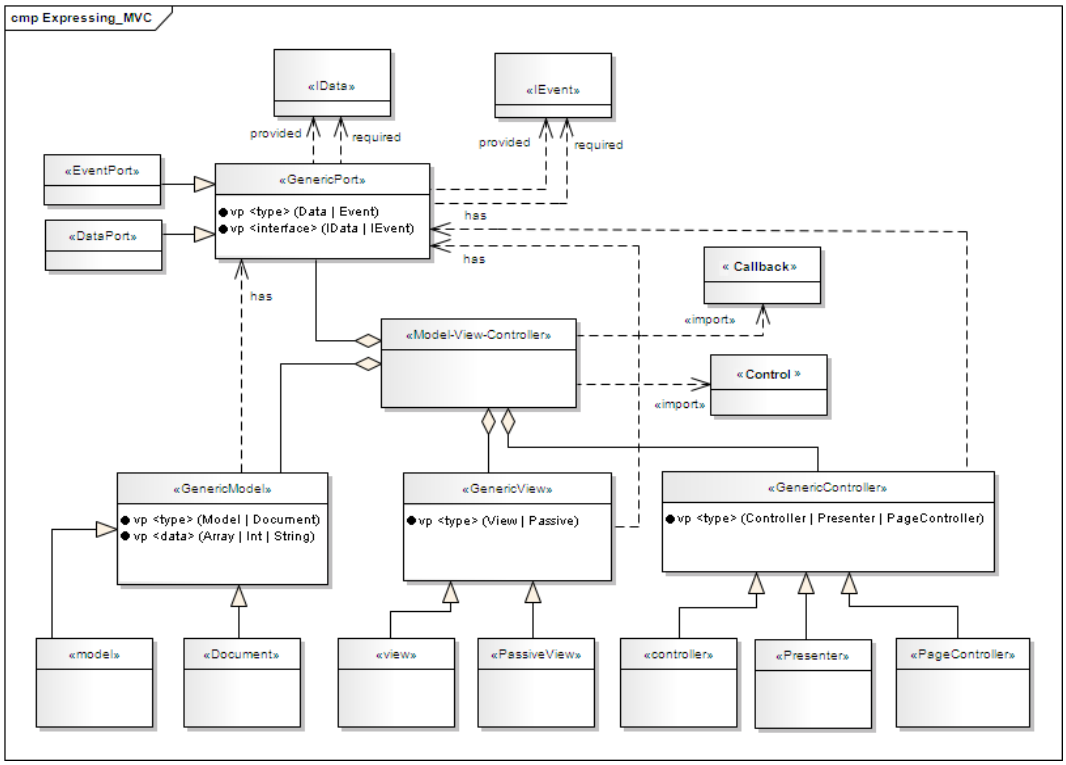

Figure 2: The Model-View-Controller defined in UML

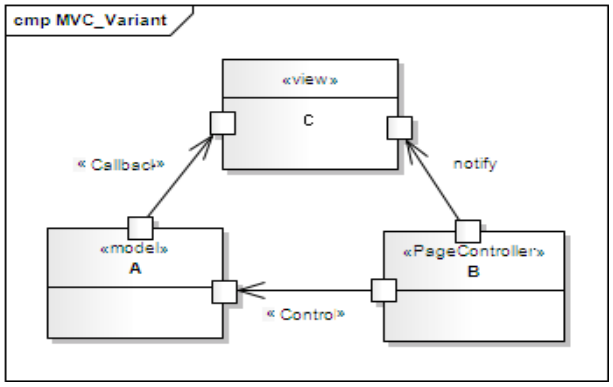

Figure 3: Modeling the PageController MVC Pattern Variant

```
 stemp :  Bag(Component) = comp.ownedPort->select(j |
  j.oclAsType(Port), self.ownedPort =
   EventPort and i.oclAsType(Port) in
if
 EventPort.getOwner() = 'GenericModel 'and
  View.ElementType = 'vp '
then
 true
else
 false
endif
```

A combined use of the specialized pattern participants and primitives results in the MVC structure as shown in figure 3:

5. CASE STUDY

The basic functionality of the IS2000 system [13] is to acquire images as raw data and convert them into sensor readings and images suitable for viewing. The system has a set of acquisition procedures aimed at customized acquisition of the images. The product requirements are expected to change during the development and future life span of the system. The current design of the system does not employ pattern modeling. However, a study of the system design documentation revealed different variants of architectural patterns modeled within its design e.g. Pipes and Filters, Layers, and Model-View-Control(MVC). In this section, we re-design parts of the system by applying two prominent architectural patterns to IS2000 system design. Due to space restriction, we provide only an excerpt of the design i.e. we cover only the Component-Connector view and leave certain sub-systems un-touched. However, with this limitation the study is not meant to be exhaustive as the same approach can be followed to express the same system design in different architectural views such as the process flow view, interaction diagrams etc.

5.1 Modeling the Pipes and Filters Pattern Variant within IS2000 System

One prominent architectural pattern used in the IS2000 system is the Pipes and Filters pattern providing a chain of image processing functions. The chain receives input as raw data and after performing a number of processing tasks produces refined data to the user and database. As shown in Figure 4, each filter in the Imaging component acts as a data processor by receiving input at one end and forwarding processed data to the next filter in the chain. The Acquisition

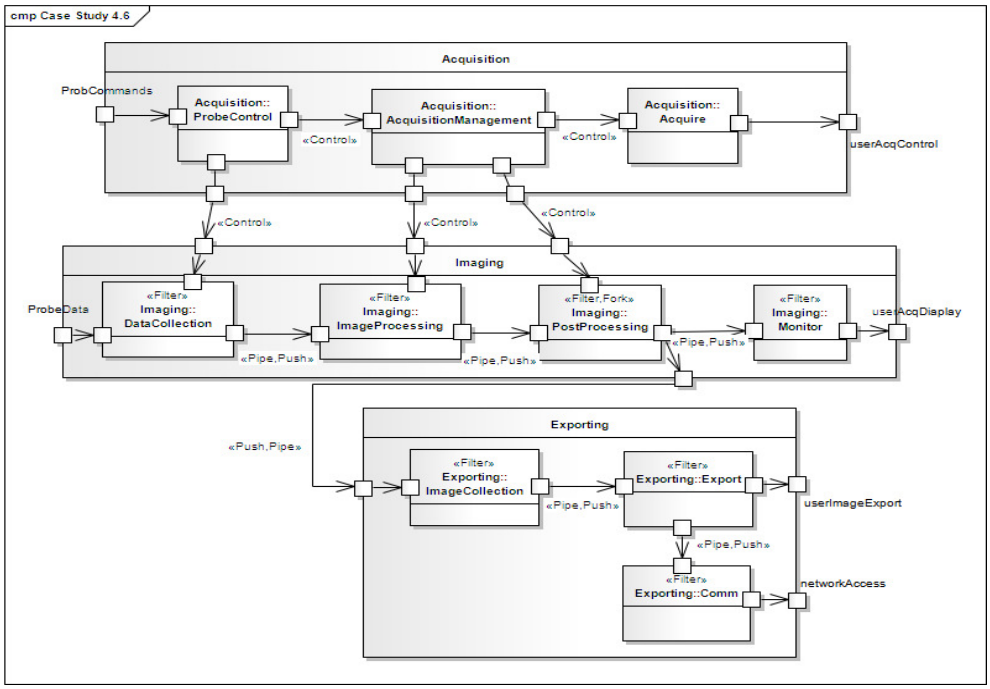

Figure 4: Pipes and Filters Pattern Variant Expressed in IS2000 system

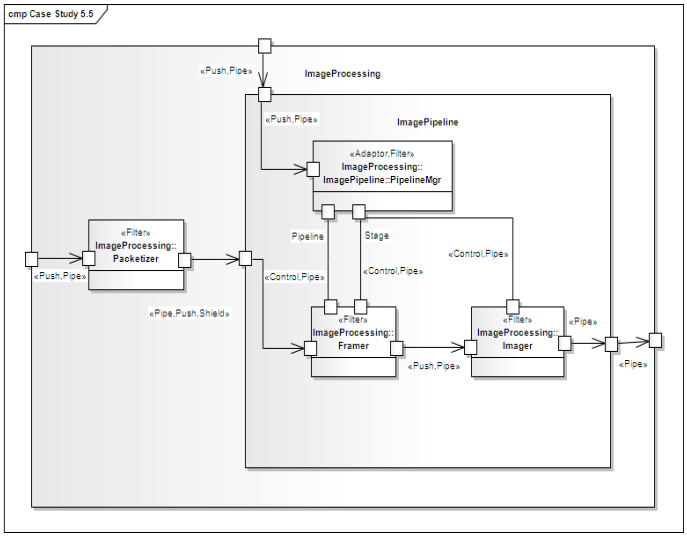

Figure 5: Internal Structure of the ImageProcessing Component

component controls the data processing, the Imaging component processes raw data, and the Exporting component sends data to other systems.

The PostProcessing component pushes data to both the Monitor and Exporting components, forming a *Fork* which is a variation from the documented Pipes and Filters pattern. In the Acquisition component, the ProbeControl, AcquisitionManagement, and Acquire components control the processing of data, which is expressed using the *Control* primitive. In the Imaging and Exporting components where the actual data processing takes place; the flow of data is expressed using the *Push* primitive. From the existing Pipes and Filters pattern elements, as defined in the previous section, we apply the *Filter* stereotype on sub-components of the Imaging and Exporting components, *Pipe* stereotype to express the flow of the image data, and *Fork* stereotype to express the pattern variant as shown in Figure 4.

Inside the ImageProcessing component, as shown in Figure 5, the PipelineMgr component transfers processing control to both the Framer and Imager component which is expressed using the *Control* primitive. Furthermore, it converts the incoming data in a suitable form to be sent to the Framer and Imager component that is expressed using the *Adaptor* primitive. In addition to applying these primitives in the design, the Pipes and Filters structure is expressed using the *Filter* and *Pipe* stereotypes.

5.2 Modeling the MVC Pattern Variant within IS2000 System

In the example IS2000 system, the GUI module is responsible for defining and managing the user display and handle user events, whereas the core functionality that defines necessary action when an event takes place is handled by the application module. The GUI module can accept input from the mouse, keyboard, or the screen menus to which the application module set up the acquisition parameters, forward messages, or report the status of acquisition to the GUI. Thus, the display and event handling are handled by the GUI module while the application logic resides in the application module. This kind of structure is known as the document-view architecture [1], which is a variant of the MVC pattern. The Document component corresponds to the Model in MVC while the View component of the Document-View merges the Control and View components. Figure 6 shows the Document-View pattern variant expressed using the *Callback*, *Control*, and *Push* primitives

in combination with the *View* and *Document* stereotypes to fully express the pattern variant.

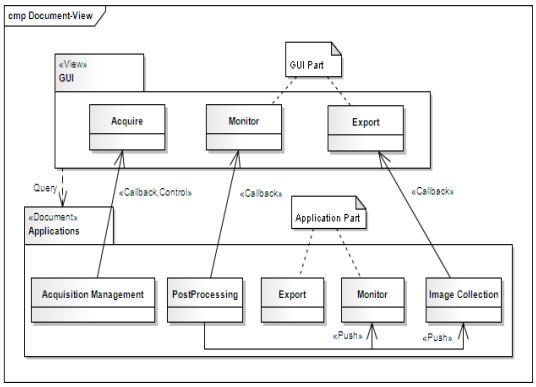

Figure 6: Modeling Document View MVC Pattern Variant

6. RELATED WORK

Using different approaches, few other researchers have been working actively for the systematic modeling of architectural patterns [14, 15]. Garlan et al. [14] proposes an object model for representing architectural designs. They characterize architectural patterns as specialization of the object models where each such specialization is built as an independent environment to be applied in a specific project. Our approach significantly differs in a way that our focus is on reusing primitives and pattern participants and only where required we extend the UML elements to capture the missing pattern variant semantics.

Mehta et al. [15] focus on the fundamental building blocks of software interaction and the way these can be composed into more complex interactions. They present a classification framework with a taxonomy of software connectors and advocate the use of the taxonomy for modeling software architecture. However, the taxonomy lacks the information to model a variety of architectural patterns rather it is focused on the basic building blocks of component-based development. Our work focuses on systematic software design using the primitives, generic and specialized pattern participants within pattern variants that provide reusable architectural building blocks to design a system.

Pattern formalization approaches are another paradigm successfully practiced in software design community for modeling patterns in a system design [16] [5]. However, the existing pattern formalization approaches are mainly focused on modeling object oriented design patterns. Few formal pattern modeling approaches address architectural patterns but the pattern solutions offered by such approaches are more applicable to specific system design rather than modeling patterns in general. Our approach offers primitives, generic and specialized pattern participants that in combination can be used to model several pattern variants rather than modeling individual pattern solutions.

Mathias [17] sketches a proposal for modeling variabilities in software families with UML extensions. He uses the standardized extension mechanism of UML to model selective variability in a software design. The system-specific model that he proposes for modeling variability offers mandatory and optional modeling elements for expressing variability in a design. Our work significantly differs as we focus on defining system-independent variation points on generic pattern participants that can be re-used or further specialized in the context of a system design at hand. Moreover, our work is explicitly focused on modeling pattern variability with pattern-specific UML profiles defined whereas his work focuses on broader system-specific UML profiles.

7. CONCLUSIONS AND FUTURE WORK

The idea to use primitives and UML models extension for software design is not novel and has been applied in different software engineering disciplines [15]. The novelty of our work lies in using the primitives in combination with the generic and specialized pattern participants for expressing several pattern variants, which has not been fully addressed before.

The combined use of the architectural primitives, generic and specialized pattern participants offers an effective way to model pattern variants in a system design. We show the applicability of our approach by successfully modeling architectural patterns variants within a system design. The scheme to use primitives in combination with the generic and specialized pattern participants offers: a) reusability support by providing vocabulary of patterns' solution participants that entail the properties of known pattern participants; b) automated model validation support by ensuring that the patterns are correctly applied in a system design; and c) explicit representation of architectural patterns in system design using UML's stereotyping scheme.

As future work, we are in the process of developing a pattern modeling tool called Primus [18], which will support modeling pattern variability, analyzing the quality attributes, architectural views synchronization (e.g. synchronizing UML diagrams in different architectural views), source code generation, etc. We plan to apply our approach to industrial case studies for designing systems using primitives, generic and specialized pattern participating. We believe that we can cover more architectural patterns in the near future, which will provide a better re-usability support to the architects for systematically expressing architectural patterns variants.

8. REFERENCES

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume1*. Wiley & Sons, 1996.

[2] Nenand Medvidovic and Richard N.Taylor. A classification and comparison framework for software architecture description languages. (Volume 26, No. 1):–, 2007.

[3] Nenand Medvidovic, David S.Rosenblum, David F.Redmiles, and Jason E.Robbins. Modeling software architectures in the unified modelong language. ACM Transactions on Software Enginnering and Methodology(Volume 11, No. 1):–, 2007.

[4] Morgan Bjorkander and Cris Kobryn. Architecting systems with uml 2.0. *IEEE Softw.*, 20(4):57–61, 2003.

[5] T. Mikkonen. Formalizing design patterns. In *Proceedings of the 20th international conference on Software engineering*, pages 115–124, Kyoto, Japan, 1998. IEEE Computer Society.

[6] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, Volume 6, No. 3:213–249, 1997.

[7] Object constraint language specification. *OMG Standard*, 1.1.

[8] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 133–146, 2005.

[9] Ahmad Waqas Kamal and Paris Avgeriou. Modeling architectural patterns' behavior using architectural primitives. *ECSA '08: Proceedings of the 2nd European conference on Software Architecture*, pages 164–179, 2008.

[10] Neil B. Harrison and Paris Avgeriou. Incorporating fault tolerance tactics in software architecture patterns. *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pages 9–18, 2008.

[11] Ahmad Waqas Kamal, Paris Avgeriou, and Uwe Zdun. Modeling variants of architectural patterns. *In Proceedings of 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008)*, pages 1–23, 2008.

[12] Diana L. Webber and Hassan Gomaa. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, 2004.

[13] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[14] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT Softw. Eng. Notes*, 19(5):175–188, 1994.

[15] Nikunj R. Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, New York, NY, USA, 2003. ACM.

[16] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Proceedings of the 26th International Conference on Software Engineering*, pages 666–675. IEEE Computer Society, 2004.

[17] Mathias Klaus. Generic modeling using uml extensions for variability. *Intershop Research Intershop, Jena Software Engineering Group, Dresden University of Technology*, 2004.

[18] Nick Kirtley, Ahmad Waqas Kamal, and Paris Avgeriou. Developing a modeling tool using eclipse. *International Workshop on Advanced Software Development Tools and Techniques, Co-located with ECOOP 2008*, 2008.

9. APPENDIX A

This section provides an overview to 14 primitives discovered during our previous work [8, 11]. Architectural primitives, as modeling participants of the architectural patterns, serve as the building units for expressing the related patterns. A detailed explanation of defining the primitives using UML extensions can be found in [8, 11]. Our original set of primitives is comprised of the following:

1. *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.
2. *Indirection:* A component receiving invocations does not handle the invocations on its own, but instead redirects them to another target component.
3. *Grouping:* Grouping represents a Whole-Part structure where one or more components work as a Whole while other components are its parts.
4. *Layering:* Layering extends the Grouping primitive, and the participating components follow certain rules, such as the restriction not to bypass lower layer components.
5. *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.
6. *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.
7. *Shield:* Shield components protect other components from direct access by the external client. The protected components can only be accessed through Shield.
8. *Typing:* Using associations custom typing models are defined with the notion of super type connectors and type connectors.
9. *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.
10. *Push-Pull:* Push and Pull 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).
11. *Virtual Callback:* In many cases the callback between components does not exist directly, rather there exist mediator components between the source and the target components, which is expressed using the Virtual Callback primitive.
12. *Adaptor:* This primitive converts the provided interface of a component into the interface the clients expect.
13. *Passive Element:* Consider an element is invoked by other elements to perform certain operations. Passive elements do not call operations on other elements.
14. *Interceder:* Sometimes certain objects in a set of objects communicate with several other objects. Interceder components are used to decouple interacting components and reduce the communication complexity.