Modeling Architectural Patterns' Behavior Using Architectural Primitives

Ahmad Waqas Kamal and Paris Avgeriou

Department of Mathematics and Computer Science
University of Groningen, The Netherlands
a.w.kamal@rug.nl, paris@cs.rug.nl

Abstract. Architectural patterns have an impact on both the structure and the behavior of a system at the architecture design level. However, it is challenging to model patterns' behavior in a systematic way because modeling languages do not provide the appropriate abstractions and because each pattern addresses a whole solution space comprised of potentially infinite solution variants. In this paper, we advocate the use of architectural primitives for systematically modeling architectural patterns in the behavioral view. These architectural primitives are found among a number of architectural patterns and serve as the basic building blocks for modeling patterns' behavior. The main contribution of this work lies in the discovery of architectural primitives, defining architectural primitives using UML, and capturing the missing pattern semantics by using UML's stereotypes.

Keywords: Architectural Patterns, Architectural Primitives, Modeling, UML.

1 Introduction

Architectural patterns provide solutions to recurring design problems that arise in a specific context [1] [2]. These patterns propose a particular structure and behavior that can be tailored to the specific needs of the problem at hand [3] [4]. The solution of an architectural pattern is a model; applying the pattern results in incorporating that model into the software architecture of a specific system. One of the most significant aspects of modeling architectural patterns is the patterns' behavior, which are mostly represented as scenarios that define the run-time actions of the patterns [4]. Such a run-time behavior is vital for the pattern implementation as it shows the way 'pattern participants' collaborate and communicate with each other to express a pattern. We use the term 'participants' to mention the modeling elements that work in association to express architectural patterns. Unfortunately, modeling architectural patterns' behavior in a systematic way remains a challenging task mostly due to the following reasons:

a) Pattern participants do not match the architectural abstractions of commonly used modeling languages.
b) Architectural patterns' behavior can potentially be modeled in infinite different ways to balance the forces related to the problem at hand.

Architecture Description Languages (ADLs) (e.g. ACME [5] or Wright [6]) and UML [7] have traditionally been used for modeling architectural patterns. Few of these languages focus specifically on modeling patterns' behavior while few others provide general architectural abstractions that can be extended to express patterns. UML is one such widely known modeling language that offers a generalized set of interaction elements to describe behavioral aspects of software architecture. However, both ADLs and UML provide only limited support for modeling patterns [8] because the architectural abstractions provided by these languages do not match the pattern participants and because they do not provide mechanisms for modeling the infinite variability of pattern behavior.

In our previous work, we have identified a set of *architectural primitives* in the Component-Connector view [9] and the Process Flow view [10]. We consider the primitives as key participants in modeling patterns and use them as the fundamental modeling elements to express a pattern in system design. These primitives offer reusable modeling abstractions that can be used for systematically modeling pattern variants. In this paper, we extend our work by focusing on architectural primitives in the behavioral view. We show how few primitives, which are already used for modeling patterns in the structural view, can be used for modeling patterns in the behavioral view as well. We illustrate our approach by presenting how the behavior of three typical architectural patterns can be modeled with the help of these new primitives. Furthermore, since primitives alone do not capture the entire semantics of the patterns, we show how to identify the missing semantics and express them through a vocabulary of pattern-specific objects and messages.

The remainder of this paper is structured as follows: in Section 2, we motivate our choice of selecting UML's collaboration diagram for modeling patterns' behavior. In Section 3, we present our approach for representing patterns and primitives as modeling abstractions using an extension of the UML. Section 4 gives detailed information of the primitives discovered during our work. In Section 5, we use primitives and a vocabulary of design elements, for modeling three selected patterns. Section 6 elaborates on related work and Section 7 discusses the future work and concludes this study.

2 The Unified Modeling Language in the Behavioral View

Although any modeling language can be used for modeling architectural primitives as long as the selected modeling language supports an extension mechanism to handle the semantics of the primitives, the UML is our choice in this work. The motivation behind the selection of UML is: a) UML is a widely known de facto modeling language; b) UML provides explicit extension mechanisms; and c) UML supports a variety of diagrams for describing the behavioral aspects of software architecture, such as Use case, Sequence, Collaboration, Statechart, and Activity. Each of these diagrams serves specific purposes to describe software design, which at times overlap with each other. These diagrams use particular UML modeling elements, which can be extended to meet the specific needs of modeling a system. In this paper, the requirements that we consider for modeling patterns' behavior are as follows:

- *Pattern elements operations:* The operations performed by pattern participants show the true essence of pattern behavior. The operation parameters, return values, and operation type should be represented in the design.
- *Relationships among pattern elements:* The relationships define the nature of interactions performed by the objects, such as the order of occurrence of the operations, multiplicity, and direction of flow etc.
- *Pattern behavior in response to user/system interaction:* Capturing the behavior of pattern participants that can explain the major dynamics of the pattern when a specific event or user/system action takes place.

Depending on the purpose, the UML supports a variety of diagrams for modeling different aspects of system behavior. A brief description of each UML diagram for modeling system behavior and their comparison to the requirements listed above is given as follows:

- *Use Case Diagrams* describe the interaction between actors – who initiate the action – and the system. The interaction is usually described using a sequence of steps. Use cases are usually defined at a higher level where the system design is considered as a black box, and emerges from the requirements used for designing the system. The use case diagrams, being at a higher level of abstraction, are not a close match to the requirements listed above because our focus lies on detail level interactions and operations among pattern participants.
- *Sequence diagrams* use objects, events, and arrows to depict scenarios by exchanging messages between objects when a specific event occurs. They usually show the execution of a typical example. Sequence diagrams are a close match to the requirements listed above as they show the sequence of operations entailed by the architectural patterns, occurrence of events to invoke specific operations, and use messages to show the interaction among pattern participants.
- *Statechart diagrams* show interactions with other objects inside or outside the system. A state shows the execution of a specific function when an event occurs. State diagrams are more focused on transition of states among objects while our focus lies on interaction among objects, which makes these diagrams a weak option for modeling patterns' behavior in context of the requirements listed above.
- *Collaboration diagrams* depict scenarios as flow of messages. Collaboration diagrams are very similar to sequence diagrams. However, an obvious difference is that collaboration diagrams show the teamwork of messages while sequence diagrams shows the stepwise execution of messages. Similar to the sequence diagrams, we consider collaboration diagrams as a close match to our work since collaboration diagrams can show the operations taking place between the pattern participants, the relationship, and occurrence of specific events.
- *Activity Diagrams* show an operation that is invoked when a specific event occurs. The activity diagram focus on using threads for the transfer of control and data among objects and hence more often used for synchronization checks [7]. These diagrams too are not a close match to the requirements listed above, as activity diagrams do not explicitly show the relationships and interactions among pattern participants.

Thus, we focus on capturing the interaction mechanism between pattern participants using either the sequence diagrams or collaboration diagrams. While sequence diagrams are more restricted to time-bound occurrence of events, the collaboration diagrams are the best choice in this work, which rely on interactions and relationships among objects in a time-independent manner. However, both types of these diagrams are comparative in nature and can be converted from one form to the other.

3 Extending UML to Represent Patterns and Primitives

UML is a widely known modeling language and is highly extensible [7]. There are two approaches for extending UML: extending the core UML metamodel or creating profiles by extending metaclasses. Our work focuses on the second approach, i.e. we create profiles specific to the individual architectural primitives. To capture the missing patterns semantics and to express the discovered architectural primitives, we extend the UML metaclasses using UML profile mechanism. That is, we define the primitives and pattern participants as extensions of existing metaclasses of UML using stereotypes and constraints as follows:

- *Stereotypes:* We use stereotypes to extend the properties of existing UML metaclasses. For instance, the Message metaclass is extended to generate a variety of primitives and specialized messages between pattern participants.
- *Constraints:* We use the Object Constraint Language (OCL) [11] to place additional semantic restrictions on extended UML elements. For instance, constraints can be defined on associations between objects, navigability, direction of communication, etc.

a. The UML 2 Metamodel

For the primitives presented in this paper, we mainly extend or use the following metaclasses of the UML 2.0 interaction metamodel to express the primitives:

- *Messages* are used to perform operations on the objects. Messages define a specific kind of communication in an interaction and connect the MessageEnds, which store references to the adjacent objects that need to be connected.
- *Interaction* provides connection between connectable elements using message ends. It uses namespace to store the sequence of operations taking place in the collaboration diagrams.
- *MessageEnd* connects the source object to the target object, where the source and target objects own the message ends.

We have also used the following UML metaclasses in order to express the constraints on UML metamodel:

- *EventOccurence* is a specialization of the MessageEnd. The message operations use the MessageEnds to send and receive events.
- *ExecutionOccurence* is represented by two event occurrences, the start event occurrence and the finish event occurrence.

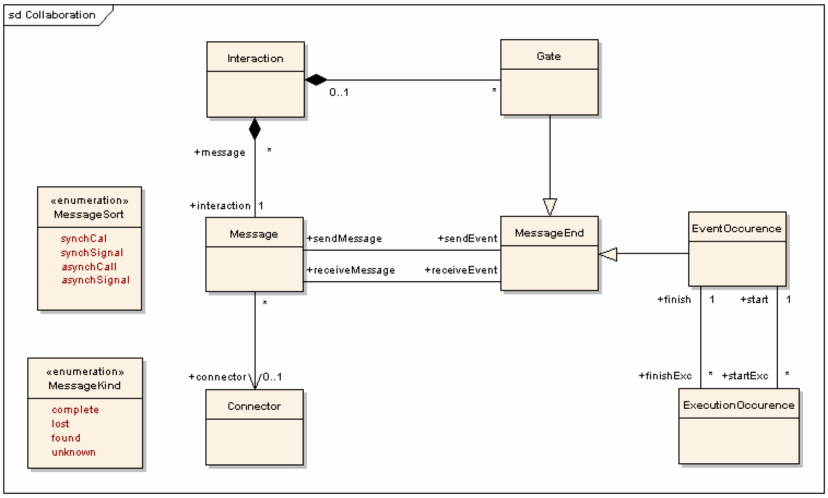

Fig. 1. Part of the UML Interaction metamodel used for defining primitives

4 Architectural Primitives

This section presents a continuation to our previous work where we have listed several architectural primitives in Component-Connector view [9] and the Process Flow view [10]. In this section, we present seven primitives discovered in the behavioral view that are repetitively found as abstractions in 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 architectural patterns' behavior, hence providing a better reusability and systematic support to model patterns. Following, we list the primitives discovered during our work and present the UML profile elements as a concrete modeling solution for expressing these primitives.

4.1 Documenting an Architectural Primitive: Push-Pull

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

Known uses in patterns

- In the Model-View-Controller [4] pattern, the model pushes data to the view, and the view can pull data from the model.
- In the PIPE-FILTER [4] pattern, filters push data, which is transmitted by the pipes to other filters and even pipes can request data from source filters (Pull) to transmit it to the target filters.
- In the PUBLISH-SUBSCRIBE [4] pattern, data is pushed from the framework to subscribers and subscribers can pull data from the framework.

- In the CLIENT-SERVER [4] pattern, data is pushed from the server to the client, and the client can send a request to pull data from the server.

Modeling Issues: Semantics of the push-pull structure is 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 above listed patters in UML is that although a Push-Pull structure is often used to transmit data among objects, it cannot be explicitly modeled using UML interaction diagrams.

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 2 illustrates these stereotypes according to the UML 2.0 interaction model.

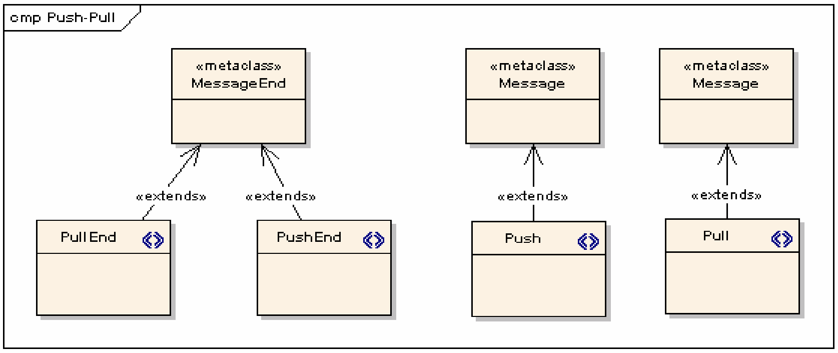

Fig. 2. UML Stereotypes For Modeling the Push-Pull Structure

<<Push>>: A stereotype that extends the 'Message' metaclass and attaches to messsage ends that connect adjacent objects.

-- A Push message has only two ends

```
inv: self.baseMessage->size() = 2
```

-- A Push message should be represented by a directed Message only

```
inv: self.baseMessage.type.MessageEnd->select(
Message = Core::MessageKind::directed).class->any(true)
```

-- The following constraint specifies the presence of interaction link between connected elements

```
inv: self.enclosingInteraction->select(
oclAsKindOf(Message)->exists(I:Interaction | I.PushEnd)
```

<<Pull>>: A stereotype that extends the 'Message' metaclass and owns Message Ends that connect adjacent objects.

-- A Pull message has only two ends

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

-- A Pull message should be represented by a directed Message only

```
inv: self.baseMessage.type.MessageEnd->select(
Message = Core::MessageKind::directed).class->any(true)
```

-- The interaction contains the message ends owned by the adjacent objects

```
inv : self.enclosingInteraction-> se-
lect(oclAsKindOf(Message)->exists(I:Interaction | I.PullEnd)
implies
select(oclAsKindOf(Message)->exists(I:Interaction |
I.PushEnd)
```

<<PullEnd>>: A stereotype that extends the MessageEnd metaclass and contains a number of operations that serve the purpose of Pull operations between connected elements.

```
inv: self.baseMessageEnd->forAll(i:Core:: MessageEnd |
PullEnd.baseMessageEnd->exists (j | j=i)
```

<<PushEnd>>: A stereotype that extends the MessageEnd metaclass and contains a number of operations that serve the purpose of Push operations between connected elements.

```
inv: self.baseMessageEnd->forAll(i:Core:: MessageEnd |
PushtEnd.baseMessageEnd->exists (j | j=i)
```

4.2 More Architectural Primitives

Due to space restrictions, we do not go into the detailed definition for the rest of the architectural primitives discovered in this work. Instead, we present a shortened modeling solution.

I. Callback

Textual Description: In a callback interaction between objects, an object B invokes an operation on object A, where object B keeps a reference to object A. Usually the callback function is invoked when a run-time event happens.

Known Uses in Patterns: MODEL-VIEW-CONTROLLER [4], OBSERVER [4], PUBLISH-SUBSCRIBE [4]

Modeling Issues: A major problem in modeling these patterns in UML is that even though callback is an active participant in the patterns, it can not be semantically represented in the interaction diagrams. A UML interaction diagram can depict the presence of a callback structure but it cannot be distinctively identified. It is hard to distinguish between many operations taking place between objects and the callback-specific operations.

Modeling Solution: To capture the semantics of callback primitive properly in UML, we use the following stereotypes: <<Callback>>, <<EventEnd>>, and <<CallbackEnd>>.

The <<Callback>> extends the Message metaclass while the <<EventEnd>> and <<CallbackEnd>> extend the MessageEnd metaclass. A callback invocation is always preceded by an event occurrence and the callee object must have subscribed itself to the caller object beforehand. In this case, the kind of message communication must be of signal type [7] where the EventOccurence takes place at the sender object (EventEnd) while the EventExecution takes place at the receiver end (CallbackEnd).

II. Forward-Request

Textual Description: Forward-Request primitives are used to depict the presence of a request forwarding mechanism. Forward-Request messages decouple the underlying system from the external objects.

Known Uses in Patterns: PEERS [2], BROKER [4], CLIENT-SERVER[9], FORWARD-RECEIVER [2], MARSHALLER [2]

Modeling Issues: A Forward-Request typically differs from simple function calls, return calls, and other forms of communications among objects. The Forwarder object decouples the underlying system implementation from external function calls and converts incoming data into matching data format without introducing further dependencies. Moreover, in certain cases, the forwarder objects can receive return values that are forwarded to the source objects. However, UML elements cannot structurally express the presence of Forward-Request operations in software design.

Modeling Solution: To capture the semantics of Forward-Request properly in UML, we propose the following new stereotypes: <<Forward-Request>>, <<ForwardEnd>>, and <<ReceiverEnd>>. The <<Forward-Request>> extends the Message class and uses the <<ForwardEnd>> and <<ReceiverEnd>> to connect the adjacent objects. Both the <<ForwardEnd>> and <<ReceiverEnd>> extend the MessageEnd metaclass and are owned by the forwarder and receiver objects respectively. To execute an operation,, the <<ForwardEnd>> invokes the sendMessage operation, which is intercepted by the receiver object using the <<ReceiverEnd>>.

III. Command

Textual Description: Calling a method in the target object typically involves invoking a specific method or procedure in the target object. The invocation operation is usually carried out on the occurrence of a specific event.

Known Uses in Patterns: MODEL-VIEW-CONTROLLER [4], PRESENTATION-ABSTRACTION-CONTROL [2], LAYERS [4]

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

Modeling Solution: To capture the semantics of Command primitive properly in UML, we propose two new stereotypes: <<Command>>, and <<CommandEnd>>. The <<Command>> extends the Message class and uses the <<CommandEnd>> to invoke command on the target object when a specific event occurs. The <<CommandEnd>> extends the MessageEnd metaclass and is owned by the command invocation object.

IV. Asynchronous Message

Textual Description: In an asynchronous communication, the message sender continues with its operation without waiting for any reply from the message receiver.

Known Uses in Patterns: PIPE-FILTER [4], CLIENT-SERVER [2], BROKER [4]

Modeling Issues: The patterns listed above often use Asynchronous messaging. UML supports the invocation of asynchronous messages when a specific event occurs. However, it does not enforce any constraints in distinctively recognizing the asynchronous operations. Various architectural patterns use degrees of asynchrony in their operations. In the most common form of asynchronous communication, the sender's data is buffered in queues without waiting for the recipient to pick the data. The current UML collaboration diagrams support the Asynchronous messaging; however, there are two major issues:

- Even though the UML diagrams have a support for Asynchronous messaging, they do not differentiate between the return values from the target objects. It is an ambiguous 'hint' to determine whether the return value is merely a notification event about the receipt of message or the actually processed data.
- Asynchronous messages are often buffered in queues until the target object notifies about its availability using events, often much later in the time. Such a structure cannot be un-ambiguously determined in UML interaction diagrams where a number of operations among objects are taking place at the same time.

Modeling Solution: We use the <<AsynchMessage>> stereotype along with the existing UML interaction diagram functions for modeling the asynchronous communication among the objects. The <<AsynchMessage>> extends the Message metaclass and uses the existing MessageSend and MessageReceive operations to guarantee that the invocation flag is active whenever an operation is invoked. We further constrain the Asynchronous communication to ensure that the method that invoked the operation is not bound to receive the results and only a notification event can inform the receipt of message.

V. Synchronous Message

Textual Description: In a synchronous communication, the sender waits till the receiver finishes the activated operation.

Known Uses in Patterns: PIPE-FILTER [4], CLIENT-SERVER [2], BROKER [4]

Modeling Issues: The patterns listed above often use Synchronous messaging. UML denotes a synchronous message with a solid arrowhead. We specify additional constraints on UML synchronous messages to provide a clear depiction of synchronous message.

Modeling Solution: We add a simple extension to the UML metamodel by proposing the <<SynchMessage>> stereotype for modeling the synchronous communication between objects. The <<SynchMessage>> extends the Message metaclass and uses the existing UML synchmessage operations to ensure that: a) a synchronous message is always represented with a directed association; b) an end-to-end connection is established with the target object, which owns the EventEnd and returns a flag each time a data processing is completed; and c) a return operation is mandatory for the synchronous communication to update the status of the operation that invoked the synchronous communication.

VI. Call-Slave

Textual Description: The objects called slaves provide sub-services on behalf of a master object. The master also keeps reference to all the slave components.

Known Uses in Patterns: MASTER-SLAVE, PRESENTATION-ABSTRACTION-CONTROLLER [2], WHOLE-PART [2]

Modeling Issues: The call-slave structure is a key participant in modeling patterns when a task is delegated to a number of sub-objects. In such a case, the dependent objects work as slaves and usually do not invoke any operations on the surrounding elements. UML interaction diagrams can depict such a structure but cannot express the semantics in the diagrams.

Modeling Solution: We propose the following stereotypes to model the Call-Slave primitive: <<CallSlave>>, <<Slave>>, and <<Master>>. The <<CallSlave>> extends the Message metaclass and provides a selfMessage operation to invoke operations that further call upon slave objects. Both the <<Slave>> and <<Master >> represent the objects with further constraints such that only the <<Master>> object can access the <<Slave>> objects.

5 Modeling Architectural Patterns Using Primitives

In this section, we use the primitives described in the previous section to model known variants of three selected architectural patterns: Pipe-Filter, Model-View-Controller (MVC) and Client-Server. As aforementioned in the introduction, primitives capture only part of the semantics of the patterns, since there are semantics specific to individual patterns and not recurring in several patterns. Therefore, in order to complete the behavioral modeling of patterns, we need to find the missing pattern semantics and express them through a stereotyping scheme. Due to space limitation, we only provide detailed OCL constraints for the Pipe-Filter, while we omit the OCL code for the MVC and Client-Server.

5.1 Pipe-Filter

The Pipe-Filter pattern consists of a chain of data processing filters, which are connected through pipes. The output of one filter is passed through pipes to the adjacent filter. The elements in the Pipe-Filter pattern can vary in the functions they perform. For instance, pipes can buffer data, form feedback loops or fork/join structures, filters can be active or passive etc. Each such function can be described with a specific scenario to depict the behavior of the pattern. The primitives discovered so far address many such variations. We select the Push, Pull, and Synchronous Message 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.
- Data is sent from one filter to the next filter in the chain using synchronous operations.

Missing Pattern Semantics: Despite the reusability support offered by the selected primitives, the Pipe-Filter pattern semantics cannot be fully expressed in design because the feedback, pipe, and filter structure are still missing. We apply the Feedback stereotype on the Push primitive to capture the presence of feedback loop in the Pipe-Filter pattern. Such a structure represents data being pushed from one filter object to another filter object using the feedback loop. The original Push primitive, as described in section 4, extends the UML metaclasses of Message and MessageEnd. The feedback stereotype further specializes the Push primitive by stereotyping it as Feedback without introducing new constraints.

<<Feedback>>: A stereotype that is applied on the Push primitive for expressing the Feedback operation in the Pipe-Filter pattern. The semantics of a feedback operation are similar to Push and Pull data streams operation.

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 Object metaclass of UML and owns message ends.

-- A Filter object owns the MessageEnds of the associated pipes such that within an interaction, it owns the receiver end of source pipe and the sender end of next pipe in the chain

```
inv: self.enclosingInteraction->
select(oclAsKindOf(Object)->exists(I:Interaction |
I.MessageOut) implies self.enclosingInteraction->
select(oclAsKindOf(Object)->exists(I:Interaction |
I.MessageIn)
```

<<MessageOut>> A stereotype that extends the MessageEnd class and owned by the filter objects

```
inv: self.enclosingInteraction->select(
oclAsKindOf(Message)->exists(I:Interaction | I.MessageOut)
```

<<MessageIn>> A stereotype that extends the MessageEnd class and owned by the filter objects

```
inv: self.enclosingInteraction->select(
oclAsKindOf(Message)->exists(I:Interaction | I.MessageIn)
```

The fifth stereotype that we use from the existing vocabulary of design elements is the 'Pipe' that is defined as follows:

<<Pipe>>: A stereotype that extends the Message metaclass of UML and attaches the MeesageEnd of source object to the MessageEnd of the target object.

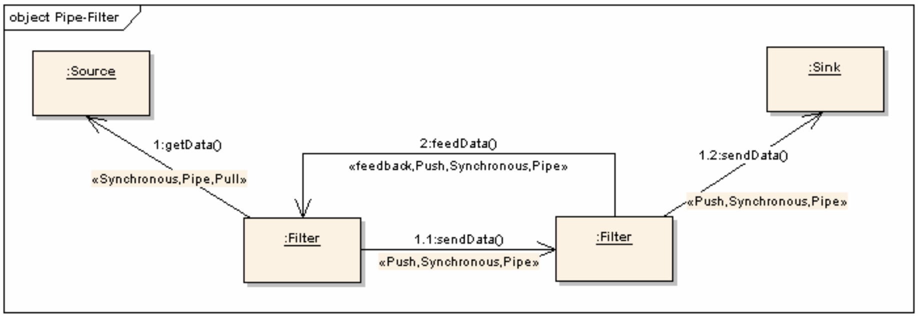

Fig. 3. Modeling Pipe-Filter Pattern Using Primitives and Design Elements

As shown in the figure above, the first filter object pulls data from the source object, and after processing pushes this data to the next filter in the chain. The second filter sends data back to the first filter using feedback pipe for further processing, and sends the final processed data to the sink.

5.2 Model-View-Controller

The behavior of MVC pattern relies on the functions performed by the following elements: Model, View, and Controller. The Model provides the functional core of the application and notifies views about data changes. Views retrieve information from the model and display it to the user. Controllers translate events into requests to perform operations on the view and model elements.

As a first step, we map the MVC pattern to the list of available primitives. We select the callback and command 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.
- Controller issues a command request on the model and view objects when some event occurs.

Missing Pattern Semantics: However, not every aspect of the MVC pattern can be modeled using the existing set of primitives. For instance, the Model, View, and Controller objects are not mapped to any primitives discovered so far. Keeping in view the general nature of these objects and their mandatory use in modeling different variants of the MVC pattern, we include the <<Model>>, <<View>> and <<Controller>> stereotypes in the existing vocabulary of pattern elements, as described below.

<<Model>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Controller and View objects.

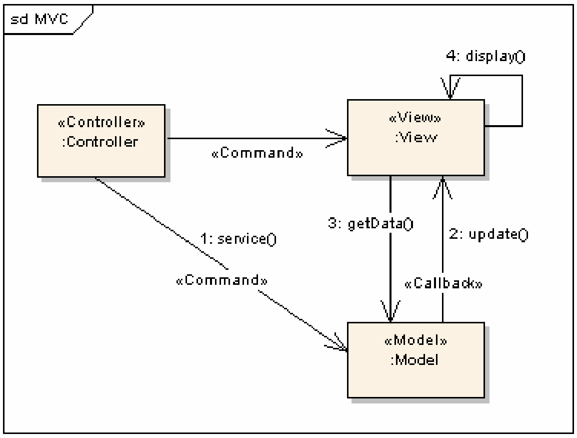

Fig. 4. Modeling the MVC Pattern Using Primitives and Design Elements

<<Controller>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Model and View objects.

<<View>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Model and Controller objects.

5.3 Client-Server

In a typical Client-Server pattern variant, the server offers operations that are accessed by the clients and even clients can perform domain-specific operations at their own. Usually a broker pattern is used to establish connections between client and server. The client sends request to the broker asking to fulfill a specific task. The broker in response looks for the appropriate server and assigns the task to the server. The server provides the functional core of the application and uses the broker to send information back to the clients.

As a first step, we map the Client-Server pattern to the list of available primitives. We select the forward-request, asynchronous, and command primitives for modeling the Client-Server pattern. The rationale behind the selection of these primitives is as follows:

- The Server issues a command request to the clients when some event occurs.
- The Client and Server side proxies synchronously forward requests to other objects.

Modeling Pattern Semantics: However, not every aspect of the Client-Server pattern can be modeled using the existing set of primitives. For instance, the Client, and the Server objects are not mapped to any primitives discovered so far. Keeping in view the general nature of these objects, we provide reusability support by making these two pattern participants available in the existing vocabulary of design elements.

<<Client>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Server and mediator objects.
<<Server>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Client, surrounding objects, and mediator objects.

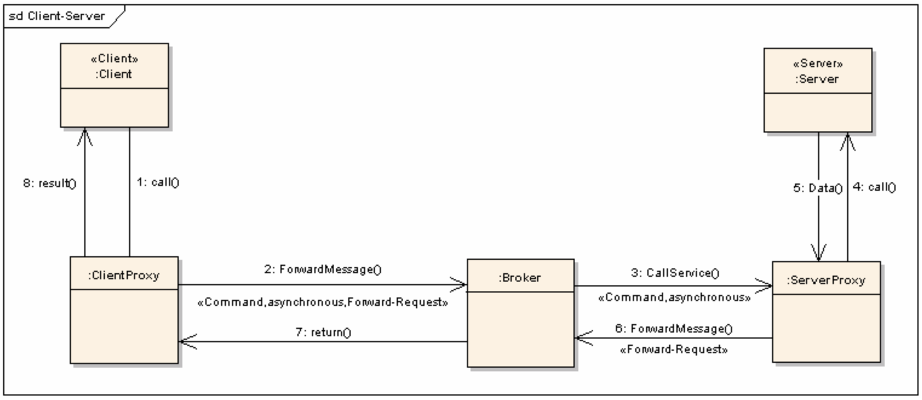

Fig. 5. Modeling the Client-Server Pattern Using Primitives and Design Elements

6 Related Work

The work described in this paper is based on our previous work [9] where we present an initial set of primitives for modeling architectural patterns in the component-connector view. However, the idea to use primitives for software design is not novel and has been applied in different software engineering disciplines [12]. The novelty of our work lies in the use of primitives for systematically modeling the behavior of architectural patterns.

Using different approaches, other researchers have been working actively on the systematic modeling of architectural patterns. Garlan et. al. [13] propose an object model for representing architectural designs. They characterize architectural patterns as a specialization of 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 as our focus lays on reusing primitives and pattern participants, which are defined as specializations of UML elements.

Werner et. al. [14] uses message sequence charts to propose a language that is capable enough to fully express the behavioral specification of systems using use cases and scenarios. Their work focuses on the execution of scenarios when different kinds of events occur for message calls of type e.g. asynchronous message, synchronous message. In our approach, we also use messages as a base for interaction but our focus revolves around modeling patterns where we use primitives and pattern participants' definitions as reusable abstractions.

7 Conclusion and Future Work

The combination of architectural primitives and the vocabulary of design elements offers a systematic way to model patterns' behavior in system design: the primitives and the design elements are reusable architectural abstractions in the form of extended UML elements; the semantics of the primitives and subsequently of the patterns can be validated by checking the OCL constraints; the patterns can be manually or automatically detected in the system design. In this paper, we have extended our existing pool of primitives with the discovery of seven more primitives in the behavioral view. Moreover, with the help of some example patterns, we demonstrated the feasibility of our approach for modeling architectural patterns using primitives.

To express the discovered primitives and design elements vocabulary, we have used UML2.0 for creating profiles. Compared to earlier versions, UML2.0 has come up with many improvements for expressing architectural elements. However, we still find UML a weak option in modeling many aspects of architectural patterns, e.g. having weak messaging support. As a solution to this problem, we regard the extension mechanism of the UML as an effective way for describing new 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 enable the user to apply *selective* profiles in the model.

As future work, we would like to advance the automation of our approach by developing a tool, which supports modeling pattern variability, documenting design decisions, analyzing the system quality attributes, consistency checking between the structural and the behavioral views, and source code generation. We believe that in different architectural views, more primitives will be discovered in the near future, which will provide a better re-usability support to the architects for systematically expressing architectural patterns.

References

[1] Avgeriou, P., Zdun, U.: Architectural Patterns Revisited - A Pattern Language. In: Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLOP), Irsee, Germany, pp. 1–39 (2005)

[2] Buschmann, F., Henney, K., Schmidt, C.D.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. John Wiley & Sons, Chichester ISBN 978-0-471-48648-0

[3] Harrison, N., Avgeriou, P.: Pattern-Driven Architectural Partitioning – Balancing Functional and Non-Functional Requirements. In: First International Workshop on Software Architecture Research and Practice (SARP 2007), Silicon Valley, USA, p. 21. IEEE, Los Alamitos (2007)

[4] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Chichester (1996)

[5] Garlan, D., Monroe, R., Wile, D.: ACME: An Architecture Description Interchange Language. In: Proceedings of CASCON 1997, Toronto, Ontario, pp. 169–183 (1997)

[6] Allen, R., Garlan, D.: A Formal Basis For Architectural Connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)

[7] Unified Modeling Language: Superstructure, version 2.0, Final Adopted Specification, ptc/03-08-02, `http://www.omg.org/cgi-bin/doc?formal/05-07-04`

[8] Kamal, A.W., Avgeriou, P.: An evaluation of ADLs on modeling patterns for software architecture design. In: 4th International Workshop on Rapid Integration of Software Engineering Techniques, Luxembourg (2007)

[9] Zdun, U., Avgeriou, P.: Modeling Architecture Patterns using Architecture Primitives. In: 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, pp. 133–146 (2005)

[10] Zdun, U., Avgeriou, P., Hentrich, C., Dustdar, S.: Architecting as Decision Making with Patterns and Primitives. In: Proceedings of the Third Workshop on Sharing and Reusing architectural Knowledge (SHARK), pp. 11–18. ACM, New York (2008)

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

[12] Mehta, N.R., Medvidovic, N.: 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, Helsinki, Finland, pp. 347–350 (2005)

[13] Garlan, D., Allen, R., Ockerbloom, J.: Exploiting Style in Architectural Design Environments. In: Proceedings of the ACM SIGSOFT 1994 Symposium on Foundations of Software Engineering, New Orleans, LA, pp. 175–188 (1994)

[14] Damm, W., Harrel, D.: LSCs: Breathing Life into Message Sequence Charts, Formal Methods in System Design. Kluwer Academy Publishers, Dordrecht (2001)