Modeling Architectural Patterns Using Architectural Primitives

Uwe Zdun
Department of Information Systems
Vienna University of Economics
Vienna, Austria
zdun@acm.org

Paris Avgeriou
Fraunhofer IPSI
Darmstadt, Germany
paris.avgeriou@ipsi.fraunhofer.de

ABSTRACT

Architectural patterns are a key point in architectural documentation. Regrettably, there is poor support for modeling architectural patterns, because the pattern elements are not directly matched by elements in modeling languages, and, at the same time, patterns support an inherent variability that is hard to model using a single modeling solution. This paper proposes tackling this problem by finding and representing architectural primitives, as the participants in the solutions that patterns convey. In particular, we examine a number of architectural patterns to discover those primitive abstractions that are common among the patterns, and at the same time demonstrate a degree of variability in each pattern. These abstractions belong in the components and connectors architectural view, though more abstractions can be found in other views. We have selected UML 2 as the language for representing these primitive abstractions as extensions of the standard UML elements. The added value of this approach is twofold: it proposes a generic and extensible approach for modeling architectural patterns by means of architectural primitives; it demonstrates an initial set of primitives that participate in several well-known architectural patterns.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.10 [**Software Engineering**]: Design

General Terms

Design, Languages

Keywords

Software patterns, architectural patterns, modeling patterns, architectural documentation, UML, OCL

1. MOTIVATION

The software architecture of a system needs to be rigorously documented in order to profit from the advantages of architecture-centric development and evolution. One of the most significant aspects of documenting software architectures is the representation of architectural patterns (also known as architectural styles[1]). In general, a pattern is a problem-solution pair in a given context. A pattern does not only document 'how' a solution solves a problem but also 'why' it is solved, i.e. the rationale behind this particular solution. Architectural patterns help to document architectural design decisions, facilitate communication between stakeholders through a common vocabulary, and assist in analyzing the quality attributes of a software system.

There are three major approaches that have been used so far for modeling architectural patterns: (a) Architecture Description Languages (ADLs) which aim at representing software architectures in general [23], (b) the Unified Modeling Language which is a generic modeling language but can be used to describe software architectures [32, 22, 4], and (c) some formal or semi-formal approaches for the formalization of pattern specification [8, 25, 37, 21]. Unfortunately, none of these approaches succeeds in effectively modeling architectural patterns for the following reasons:

- They are too limited in the abstractions they propose, to grasp the rich concepts found in patterns. UML, to start with, falls short in offering certain standard concepts of architectural patterns [1, 22, 17]. For example in the 'pipes and filters' architectural pattern [36, 6], a pipe does not match the UML connector, since the latter cannot have an associated state or even interfaces. Furthermore there are no elements in UML to model architectural configurations such as a virtual machine [36], a blackboard [3], or a C2 topology [22]. In contrast, many ADLs inherently support a few specific patterns such as C2 [22] or *pipes and filters* [36, 6], or can be extended to represent patterns (e.g. using style repositories [26]). But except for these few patterns, ADLs cannot model the rest of the patterns. Similarly, the third aforementioned approach is basically concerned with just a handful of patterns from [11].
- They do not deal with the inherent variability of architectural patterns. This is not restricted to architectural patterns but it is a general problem of specifying patterns of any kind, because each pattern covers not only one (parametric) solution, but informally describes a whole solution space for a recurring design problem. The problem is obvious in UML and ADLs, and even more so in the third aforementioned approach that deals with the formal specification of design patterns: such methods are capable of specifying one particular solution in the solution space of the pattern, but fail to

[1] In this paper we do not distinguish between the terms 'architectural pattern' (used e.g. in [6, 33, 38]) and 'architectural style' (used e.g. in [36]). For the sake of simplicity, we shall use only the term 'architectural pattern' for the rest of this paper. Their commonalities and differences are elaborated in [2].

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

specify the whole solution space expressed by the informal pattern description.

We propose to remedy the problem of modeling architectural patterns through identifying and representing a number of 'architectural primitives' that can act as the participants in the solution that patterns convey. We use the term 'primitive' because they are the fundamental modeling elements in representing a pattern and also because they are the smallest units that makes sense at the architectural level of abstraction (e.g. specialized components, connectors, ports, interfaces). Our approach relies on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns [24]. These primitives are common among the different patterns even if their semantics demonstrate slight variations from pattern to pattern. We have 'mined' a number of architectural patterns and discovered several architectural primitives that we believe are key concepts in modeling architectural patterns and subsequently software architectures in general. We provide a modeling abstraction for each type of elicited architectural primitive, and then demonstrate that it is possible to model architectural patterns explicitly, precisely, and intuitively, through a case study. It is noted that the set of primitives identified in this paper is not exhaustive, but does contain some of the most common primitives found in popular architectural patterns.

Our general approach to define architectural primitives can take advantage of any modeling language, as long as the language can be extended to provide the syntax and semantics of the primitives. We have chosen the Unified Modeling Language for this purpose, because it has become the 'lingua franca' of software design and is vastly supported by tools. We have specified an extension of a UML 2.0 metaclass for each elicited primitive, using the standard UML extension mechanisms: stereotypes, tag definitions, and constraints. We have also used the Object Constraint Language (OCL) to formalize the constraints and provide more precise semantics of the primitives. The result is a UML profile that can be imported in modeling tools; in our case we specified the profile in Eclipse/Octopus.

The rest of the paper is structured as follows: in Section 2 we give an overview of the proposed approach. Section 3 presents the UML extension mechanism of 'Profiles' and the subset of the UML 2.0 metamodel that was used for specifying our Profile. Section 4 elaborates on the results of the approach by mining several architectural patterns and providing a full discussion for one exemplary architectural primitive and summaries for other primitives. Section 5 demonstrates the approach through a case study, and Section 6 discusses related work in this field. Finally, Section 7 sums up with conclusions and future work.

2. THE PROPOSED APPROACH

The underlying idea behind our approach is that the various architectural patterns share some common architectural 'primitives'. Thus we use the patterns of a particular architectural view, as a foundation to elicit the significant architectural primitives for that view. Specifically, we propose the following approach:

1. Analyze the architectural patterns of a given architectural view to discover common participants in their solutions. These should be recurring and probably varying instances of the same architectural concept, e.g. a special-purpose component or connector. Patterns (a) capture the variations of a solution and (b) describe the solution in a realization-independent way. For instance, pattern descriptions contain pattern variants, implementation hints, design alternatives, consequences, forces that govern a solution, and so forth. These are all sources for eliciting the architectural primitives.

2. Map the primitives to well-established architectural abstractions, like components, connectors, or ports that are a close semantic match to the primitives.

3. Represent these architectural abstractions as (meta-)modeling elements that can be used by architects. Any modeling language can potentially be extended for this purpose; we have chosen to extend UML. After finding the appropriate UML metaclasses, we define the semantics of these primitives more precisely with the help of OCL in order to facilitate the unambiguous and consistent modeling of patterns.

4. Use the derived UML extensions of primitives to model pattern instances in real case studies and validate the effectiveness of the primitives to unambiguously model architectural patterns.

It is noted that the pool of architectural patterns, we used to elicit primitives, includes some patterns that are described as 'design patterns' in the literature. In general it is hard to draw the line between architectural patterns and design patterns. In fact, it depends heavily on the viewpoint of the designer or architect whether a specific pattern is categorized as an architectural pattern or a design pattern. Consider for instance, a classical design pattern, the INTERPRETER [11]. The description in [11] presents it as a concrete design guideline. Yet, instances of the pattern are often seen as a central elements in the architecture of software systems because an INTERPRETER is a central, externally visible component. Therefore the pattern is treated as an architectural pattern (see [36]). Thus we refer to such design patterns as architectural patterns, considering them at an architectural level of abstraction.

3. EXTENDING UML TO REPRESENT THE PRIMITIVES

3.1 A UML profile

According to the UML 2.0 standard [28] there are two ways to extend the language: the *hard extension* produces an extension of the language meta-model, i.e. a new member of the UML family of languages is specified; the *soft extension* results in a *profile*, which is a set of stereotypes, tag definitions, and constraints that are based on existing UML elements with some extra semantics according to a specific domain. In order to model the architectural primitives we have chosen the *soft* extension mechanism of UML (i.e. the definition of a profile for architectural primitives) for the following reasons:

- A UML profile is good enough for this task since there are already existing UML metaclasses that are semantically a close match to the architectural primitives. Therefore we can simply extend the semantics of these metaclasses rather than having to define completely new metaclasses.

- The users of this profile will feel comfortable by using stereotypes that are extensions of existing metaclasses rather than using concepts they are not familiar with. The learning curve can thus be minimized.

- A profile is still valid, standard UML, so we can count on support from the existing UML tools, rather than offer proprietary UML tools which are rarely used in practice.

We also use OCL to define the necessary constraints for the defined stereotypes so as to (semi-)formalize their semantics. OCL constraints are the primary mechanism for traversing UML models and specifying precise semantics on metaclasses and stereotypes.

3.2 The UML 2 metamodel

This section briefly presents the existing UML 2.0 metamodel for architectural description, and in particular those metaclasses that we have extended in order to model the architectural primitives. It is noted that, according to the software architecture community, an architectural description is comprised of multiple views [7, 15, 16, 20]. In this paper we focus on the view that is considered to contain the most significant architectural information, which is the *component-and-connector* view [7]. This view deals with the components, which are units of runtime computation or data-storage, and the connectors which are the interaction mechanisms between components [29, 7]. We have focused on this view because the patterns that we have mined concern mainly this view. However other architectural patterns from other views, such as the 'logical' or 'module' view, can also be searched for primitives, as will be stated in Section 7.

The following UML 2.0 metaclasses are extended to model architectural primitives in the component and connector view, mainly taken from the composite structures and components packages:

1. *Components* are specializations of classes and therefore have attributes and operations, but are also associated with provided and required interfaces. They inherit indirectly from EncapsulatedClassifier and thus may own ports that formalize their interactions points.

2. *Interfaces* serve as contracts that components must comply with. An interface is either a *provided interface* that describes a set of functionalities offered by a component, or a *required interface* that describes a set of functionalities that a component expects from its environment.

3. *Ports* specify a distinct interaction point between the component that owns the port and its environment, or between the component and its internal parts (properties). Ports may specify required and provided interfaces of the component that owns them.

4. *Connectors* are either *assembly connectors* that connect the required interface of one component to the provided interface of a second, or *delegation connectors* that link the ports of a component to its internal parts.

5. *Packages* are mechanisms for grouping model elements either by owning them or importing them. They also provide a namespace for uniquely identifying the elements by their name.

We have also used the following UML metaclasses in order to express the OCL constraints while traversing the UML metamodel: AggregationKind, Association, Classifier, ConnectableElement, ConnectorEnd, EncapsulatedClassifier, Feature, RedefinableElement, Namespace, NamedElement, PackageableElement, Property, RedefinableElement, VisibilityKind. Finally, it is noted that UML 2.0 provides the means to describe a design pattern through the Collaboration metaclass, as an interaction between instances of components and connectors. However we do not use this metaclass since it is also bounded by the limitations for modeling patterns discussed in Section 1.

The specification of the primitives was implemented with the help of the Octopus plug-in (http://www.klasse.nl/) in the Eclipse environment (http://www.eclipse.org/). We chose this tool for specifying the primitives because Eclipse is open-source and widely used, and also because the Octopus plug-in can statically check OCL 2.0 constraints. For all OCL constraints we assume the standard UML 2.0 role names for the extensions: "baseX", where X is the extended metaclass, and "extensionY", where Y is the stereotype name.

Figure 1 illustrates part of the UML metamodel that contains the aforementioned metaclasses and shows their relationships, especially for traversing OCL constraints. The figure has been adapted from the UML 2 standard [28], and, for simplicity, some details have been omitted.

4. MODELING ARCHITECTURAL PRIMITIVES

In this section, we provide more details about our approach, demonstrating the elicitation of architectural primitives from general purpose architectural patterns, and modeling them with a UML 2.0 profile. First we will demonstrate how to document architectural primitives. Next we present, as an example, the full documentation of the Callback primitive. In the last subsection, a number of other primitives are presented in abbreviated form.

4.1 Template for architectural primitive documentation

Once an architectural primitive is elicited it needs to be documented. We propose a simple template consisting of four elements:

- *Textual description*: A textual description and discussion of the architectural primitive.
- *Known uses in patterns*: A short description of the patterns in which the architectural primitive participates.
- *Modeling issues*: An explanation why this primitive cannot be modeled with standard UML and thus needs to be supported with a UML extension.
- *Modeling solution*: A description of UML 2.0 extensions, containing stereotypes, possibly with tag definitions, and constraints.

4.2 Example of an architectural primitive documentation: Callback

Textual description:

'Callback' is described as follows:

> A callback denotes an invocation to a component B that is stored as an invocation reference in a component A. The callback invocation is executed later, upon a specified set of runtime events, usually implemented as methods. Between two components A and B, a set of callbacks can be defined, also usually implemented as methods. Note that in this description A might be equal to B. In essence, the callbacks between two components A and B are a set of tuples. Each tuple consists of one method $methodA_x \in Methods_A$ that represents a trigger event and a method $methodB_x \in Methods_B$ that is a callback, like:
>
> $$\begin{aligned} Callbacks_{AB} \quad = \quad & \{(methodA_2, methodB_1), \\ & (methodA_1, methodB_2), \\ & (methodA_2, methodB_3), \\ & \ldots\} \end{aligned}$$

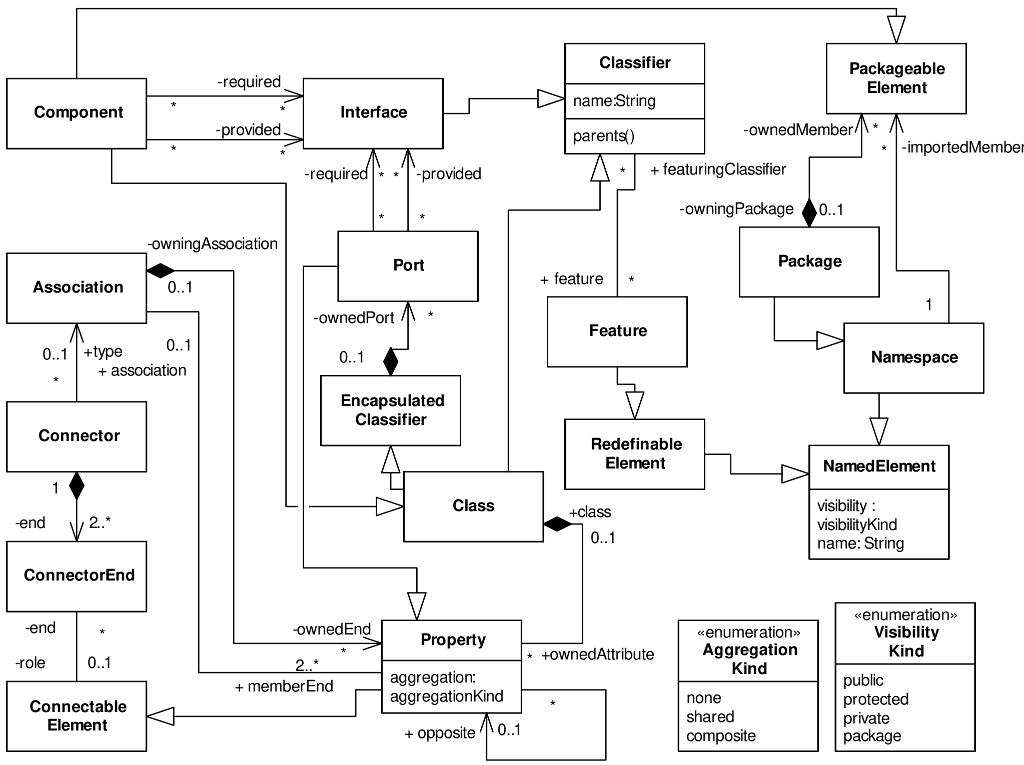

Figure 1: Part of the UML 2.0 metamodel that was used for the stereotype definition

There are two main variants of callbacks:

- The runtime events are ordinary method invocations, field accesses, or other events in the program flow. (Note that these are also called 'joinpoints' in aspect-oriented programming [19]).
- The runtime events are 'real events' in an event-based programming system, triggered by some event loop.

With regard to modeling the callback, the two variants make no difference: structurally, both kinds of callbacks are realized in the same way. Sometimes a callback has only one associated runtime event (e.g. a set with only one tuple), sometimes it is raised by a number of different runtime events.

Known uses in patterns:

Callbacks are key participants in several architectural patterns, such as the following:

- In the OBSERVER pattern [11] an observer component is notified by one or more subjects about state changes and other events. Usually the notification is realized as a callback.
- MODEL-VIEW-CONTROLLER [6] uses callbacks to inform views (and possibly controllers) about changes in the model, much like the logic behind the OBSERVER pattern.
- A REACTOR [33] is a special kind of OBSERVER that is informed about network events using callbacks.
- In the EVENT SYSTEMS pattern [36] components may broadcast a number of events. Another component may register an interest in an event by associating a callback with the event. When an event occurs, the EVENT SYSTEM dispatches all the callbacks associated with the event.
- There are various patterns describing interception architectures, such as INTERCEPTOR [33], MESSAGE INTERCEPTOR [39], and INVOCATION INTERCEPTOR [38]. Interceptors are invoked as extensions to some other invocation; thus they must be invoked, when this other invocation takes place. Usually, the interceptors are triggered by callback events like 'invocation arrived' or 'invocation finished'.
- VISITORS [11] are used to define an interpretation mechanism apart from the structure to be interpreted. They are usually called back, by the elements to be visited.

Modeling issues:

A major problem in modeling these patterns in UML is that, even though the callback-structure is a key participant in the patterns, it cannot be explicitly modeled and made visible in UML diagrams, such as component diagrams, class diagrams, or sequence diagrams. There are only some 'hints' that might imply the presence of a callback but there can be much ambiguity that could lead to false detections of callbacks. Consider the following examples of such 'hints':

- A structural indicator for a callback that we could include in UML's structural diagrams is to have a class or a component A store a reference to a method of B. Using this indicator, however, is problematic because there is no unambiguous indication whether the method reference is intended for being used as a callback or not. To make matters worse, invocation references are not necessarily realized by using a reference to a method. Many programming languages don't require a reference to the callback operation at all. For instance, in Java it is sufficient to have the operation name stored in a string to be able to look-up the operation using reflection. When the pattern COMMAND [11] is used, the callback can be encapsulated in the COMMAND. In both cases, the intended use of these structures as callbacks is not directly visible in a UML model.

- Another structural hint for callbacks is their return type. In event-driven applications, the return type of a callback is usually "void", because the callback is raised by an event, and thus the callback cannot return anything. However, this is not always the case: for instance, an interceptor often returns an error state to indicate to the interceptor architecture, whether the interceptor invocation was successful or not. Also, in non-event-driven applications, for instance, in the VISITOR and OBSERVER patterns, this rule-of-thumb does not hold: the callback may well be used with a return value.
- In some cases, where the callback can be modeled as simple recursive invocations (as in the VISITOR pattern), we can get around this problem by using an accompanying sequence diagram that shows the recursive callback (e.g. class A calls B and then B calls A back). However, there are two basic problems with this approach:
  - *No semantic annotation:* Even though the sequence diagram has a callback-like structure, the same kind of sequence diagram might be used for a 'normal' invocation going back and forth, which is not a callback.
  - *Temporal decoupling:* Callbacks are usually stored until an event happens, often much later in time, and then they are invoked upon that event. This cannot be easily depicted with a sequence diagram because of the many invocations that happen between performing the callback and the event that caused it to be invoked.

In summary, UML elements can be used as an indicator that a callback is used, but the callback structure cannot be identified unambiguously in UML's structural and interaction diagrams. Thus, the runtime behavior and interaction semantics of the callback-structure cannot be properly modeled in standard UML.

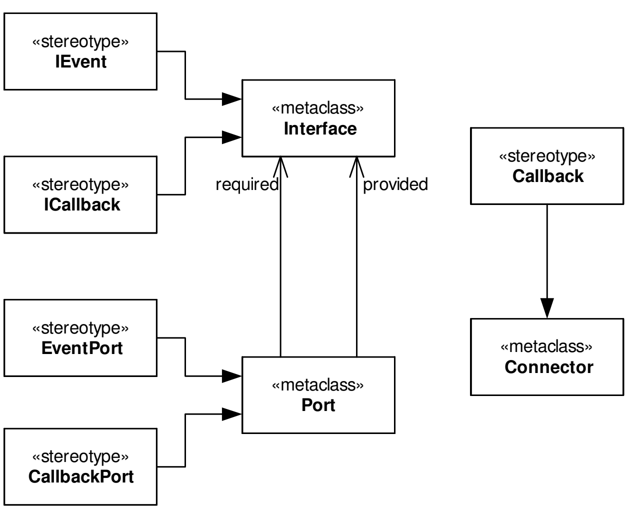

Figure 2: Stereotypes for modeling Callback

Modeling solution:

To capture the semantics of callbacks properly in UML and tackle the above problems, we propose five new stereotypes:

- ≪*IEvent*≫: A stereotype that extends the 'Interface' metaclass and contains a number of methods that are exclusively trigger events for a callback.
- ≪*ICallback*≫: A stereotype that extends the 'Interface' metaclass and contains a number of methods that serve exclusively as callback methods.
- ≪*EventPort*≫: A stereotype that extends the 'Port' metaclass and is typed by two interfaces: IEvent as a *provided* interface and ICallback as a *required* interface. This can be formalized using two OCL constraints for EventPort:

```
-- An event port is typed by IEvent as a
-- provided interface.

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


-- And: an event port is typed by ICallback
-- as a required interface.

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

- ≪*CallbackPort*≫: A stereotype that extends the 'Port' metaclass and is typed by two interfaces: ICallback as a *provided* interface and IEvent as a *required* interface. This can be formalized using two OCL constraints for CallbackPort:

```
-- A callback port is typed by ICallback as a
-- provided interface.

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


-- And: a callback port is typed by IEvent
-- as a required interface.

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

- ≪*Callback*≫: A stereotype that extends the 'Connector' metaclass and specifies the semantics of a callback connector, which connects an EventPort of a component to a matching CallbackPort of another component. This can be formalized using two OCL constraints:

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

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


-- A Callback connector connects an EventPort
-- of a component to a matching CallbackPort of
-- another component. An EventPort matches a
-- CallbackPort if the provided IEvent interface
-- of the former matches the required IEvent
-- interface of the latter, and the required
-- ICallback interface of the former matches
```

```
-- the provided ICallback interface of the latter:

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

Figure 2 illustrates these stereotypes according to the UML 2.0 Profiles package, while Figure 3 depicts the notation used for the stereotypes. All stereotypes use the notation of the metaclass they extend, adorned by the name of the stereotype in guillemets.

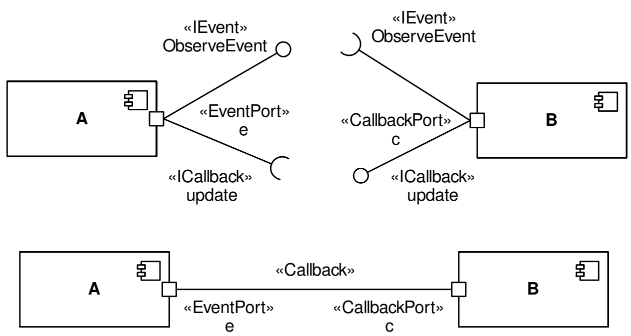

Figure 3: The notation of the stereotypes in Callback modeling

4.3 More architectural primitives

Due to space restrictions, we will not go into detail for the rest of the architectural primitives we have elicited, but we will give a brief overview of these primitives. We also consider that the readers are familiar with the details of the patterns that are referenced in the known uses. In Section 5 we will give examples for all of the primitives mentioned here. It is stressed that this list of primitives is not exhaustive. On the contrary, an analysis of more patterns and especially in other specific domains, can elicit a plethora of architectural primitives using the general approach proposed in this paper. We consider this as future work (see Section 7).

Indirection

- *Textual description*: Indirection happens when a "proxy" component receives a message on behalf of a "target" component and forwards the message to that "target", perhaps after some computation has taken place. Afterward the result is sent back through the "proxy" component again.

- *Known uses in patterns*: INDIRECTION LAYER [39], LAYERS [6], VIRTUAL MACHINE [36], INTERPRETER [36], ADAPTER [11], FACADE [11], PROXY [11], COMPONENT WRAPPER [41], WRAPPER FACADE [33], MESSAGE REDIRECTOR [39], CLIENT PROXY [38].

- *Modeling issues*: The indirection structure is not explicit in structural UML diagrams. The semantics is missing: is it an ordinary collaboration or an indirection? As in callback, sequence diagrams may help, but are ambiguous.

- *Modeling solution*: We define two stereotypes ≪*IIndirector*≫ and ≪*ITarget*≫ as extensions of the Interface metaclass. We define the stereotype ≪*IndirectionTargetPort*≫ as extension of the Port metaclass. The ≪*IndirectionTargetPort*≫ is attached to the 'target' component, and provides an ≪*ITarget*≫ interface, in order to accept requests from the 'proxy'. We define the stereotype ≪*IndirectionPort*≫ as extension of the Port metaclass. The ≪*IndirectionPort*≫ is attached to a "proxy" component, requires an ≪*ITarget*≫ interface and provides an ≪*IIndirector*≫ interface. The client of the target component can connect via the ≪*IIndirector*≫ interface to the "proxy" component, which forwards the request to the target component through its ≪*ITarget*≫ interface. The two ports are connected through a stereotyped connector, the ≪*Indirection*≫.

Grouping

- *Textual description*: A group member is part of a whole, and the whole is an abstract or virtual entity. That is, there is no component in the software architecture for representing the group as a whole, but it is made only of its parts.

- *Known uses in patterns*: Subsystem of FACADE [11], LAYERS [6], components of a BROKER [6].

- *Modeling issues*: UML's aggregation and composition relationships can be used to model part-whole relationships. But the whole should not really exist as a tangible component, it is only the sum of its parts. For instance, a subsystem contains subsystem elements, but usually there is no explicit component for representing the subsystem as a whole. Also, a UML package can be used to depict such a group, but a package may own the elements, which means that a destruction of the package would also destroy the elements. On the contrary we need a more loose relationship between the group and its members.

- *Modeling solution*: We add a simple extension to the UML metamodel for modeling groups: a stereotype ≪*Group*≫, extending the Package metaclass, is used to model a group, providing a namespace for the different group member components. We constrain the Group stereotype, so that only components can be its members, and these components are only imported and not owned by the group.

Layering

- *Textual description*: Layering builds upon the Grouping primitive and further constrains it. Specifically, it entails that group members from layer X may call into layer $X-1$ and components outside the layers, but not into layer $X-2$ and below.

- *Known uses in patterns*: LAYERS [6], LAYERED SYSTEM [36], OBJECT SYSTEM LAYER [41], INDIRECTION LAYER [39].

- *Modeling issues*: The problems in modeling Layering are similar to Grouping, but additionally we need to ensure that calls between components residing in different layers do not violate the aforementioned constraints, and in contrast to groups, one layer member cannot be part of multiple layers.

- *Modeling solution*: We introduce the ≪*Layer*≫ stereotype, which specializes the ≪*Group*≫ stereotype introduced above (which itself is an extension of the Package metaclass). We also impose the following constraints: a component can only be member of one layer and not multiple layers; components who are members of layer X may call their fellow components in layer X, as well as components in layer $X-1$ but not in other layers (e.g. $X-2$ and below). It is noted that there is no constraint about calling components in layer $X+1$ or above, since this is a specific issue to the pattern realization. Also, we introduce the tag definition layerNumber for Layers which represents the number of the layer in the ordered structure of layers.

Aggregation Cascade

- *Textual description*: A COMPOSITE [11] describes part-whole hierarchies where a composite object is composed of numerous subparts. Both composite and leaf components inherit from the same class and are treated uniformly by clients. For example a GUI widget can call its parts to paint themselves, and they call their parts and so on. A cascade [9] is a COMPOSITE structure with (recursive) constraints of the form: "a composite A can only aggregate components of type B, B only C, etc".

- *Known uses in patterns*: COMPOSITES [11] with constraints, CASCADE [9], ORGANIZATION HIERARCHY [10].

- *Modeling issues*: For this primitive, we could consider the UML Aggregation, which is a special form of the UML Association. Through Aggregation, a whole aggregates parts, and a part cannot contain its whole, but it is possible for a part to be aggregated in multiple wholes. That is, links between hierarchies are possible, but not circular links. In our primitive, the composites call their parts recursively, and there are recursive composition constraints. UML's aggregation cannot perform such recursive calls or 'cascading' constraints.

- *Modeling solution*: We constrain all components of the hierarchy, composites and leafs, to inherit from the same component type. Furthermore we define a stereotype ≪*AggregationCascade*≫ as an extension of the stereotype ≪*Indirection*≫, which itself extends the Connector metaclass. An Aggregation Cascade connects a composite to its parts. It extends Indirection since it forwards the recursive operations to clients. Since it specializes Indirection all the constraints from Indirection are also valid here. The Association that types the Connector is an Aggregation, to enforce that this is really a connector between a composite and its parts. Since we introduce the aggregation between two specific, connected components, and not between a Composite and a generic interface (as in the COMPOSITE pattern), these aggregations are constrained so that "A composite A can only aggregate components of type B, B only C, etc".

Composition Cascade

- *Textual description*: A Composition Cascade builds upon Aggregation Cascade, and further enforces that a component may not be part of more than one composite at any time. In this case, composites have a lifecycle responsibility for their parts. That is, the whole may take direct responsibility for creating or destroying the parts, or it may accept an already existing part, and later pass it on to some other whole that assumes responsibility for it. Again, these lifecycle operations need to be applied in a recursive fashion: e.g. a composite that is destroyed, destroys its parts, which recursively destroy their parts and so on.

- *Known uses in patterns*: Same as Aggregation Cascade.

- *Modeling issues*: We face the same modeling issues as in Aggregation Cascade, but we need to model a more rigid aggregation relationship: a component may not be part of more than one composite at any time. The recursive operations must also include the aforementioned lifecycle operations.

- *Modeling solution*: The solution is to constrain even more the ≪*Aggregation Cascade*≫ Connector. We thus define the ≪*CompositionCascade*≫ stereotype as an specialization of ≪*AggregationCascade*≫. In this case the Association that types the connector is a Composite Aggregation, so each part can only be owned by one Composite.

Shield

- *Textual description*: One or more components act as 'shields' for a set of components that form a subsystem. No client should be allowed to access members of the subsystem directly, but access should happen only through these 'shields'.

- *Known uses in patterns*: FACADE as a public subsystem interface [11], MESSAGE REDIRECTOR [39], OBJECT SYSTEM LAYER [41], LAYERS [6], remoting patterns used in a layered BROKER architecture [38].

- *Modeling issues*: We need to model the members of the subsystem, as well as the components shielding the subsystem. We need to make sure that no invocation can bypass the 'shield' components. These concepts cannot be represented in standard UML.

- *Modeling solution*: We utilize the Grouping primitive (or extensions of it such as Layering), described above to model the membership of the components in the 'shielded' group. We introduce the stereotype ≪*IShield*≫ that extends the Interface metaclass. ≪*IShield*≫ is offered by the components that shield the subsystem and provide access to the rest of the group members. We also introduce the stereotype ≪*Shield*≫ that extends the Connector metaclass. A ≪*Shield*≫ connector can be used by a client to connect to the "shield" component. Thus we constrain ≪*Shield*≫ to match the provided ≪*IShield*≫ interface of a "shield" component to the matching required interface of a client component. Finally, we introduce the stereotype ≪*ShieldPort*≫ that extends the Port metaclass. A port stereotyped as ≪*ShieldPort*≫ provides at least one ≪*IShield*≫ interface. ≪*ShieldPort*≫ is also extended by a tag definition, shieldGroup, for denoting the group which is shielded. ≪*ShieldPort*≫ is constrained so that all components that connect to its port and are not client components, should be members of the shieldGroup. Finally, each such component that is not itself a shield component for the same or other groups, should have a "package" visibility for all its provided interfaces. That means, the member components of the shielded group can only be accessed by other members of the group or via the ≪*IShield*≫ interfaces.

Typing

- *Textual description*: In many situations, the typing relation provided by the design or programming language is not sufficient for modeling domain types. For instance, the domain might require dynamic or constrained type dependencies. Consider for example a typical business situation: there are different Party Types in a company (e.g. "manager", "implementation group"), and a particular business entity (e.g. John, group X) can change its Party Type at runtime. For instance, a person of party type "manager" can become "senior manager", a group of type "test group" can become "implementation group", and so forth. There are usually constraints on these type changes (e.g. a group cannot take a Party Type that needs to be fulfilled by a person). In other words, a custom, dynamic type system for Party Types needs to be implemented from scratch by the developers.

- *Known uses in patterns*: TYPE OBJECT [18], KNOWLEDGE LEVEL [10], OBJECT SYSTEM LAYER [41].

- *Modeling issues*: Such typing relationships can only be modeled using UML associations. But, they are nothing more than ordinary associations, lacking the semantics of typing, such as type compliance rules, type conversion rules, inheritance, etc. Constraints of the typing relation are also not documented as such.

- *Modeling solution*: We introduce two stereotypes that extend the Connector metaclass and realize the typing relationships: ≪*TypeConnector*≫ realizes the typing relationship and ≪*SupertypeConnector*≫ realizes the supertype relationship. Both connectors are constrained not to form cycles (e.g. "A is of type B, B of type C, and C of type A" should not be allowed). Using these connectors we can document a custom-built type system. For instance, in the example above we can associate the business entity 'John' with a ≪*TypeConnector*≫ to a specific Party Type "manager", which itself has a ≪*SupertypeConnector*≫ to a generic "party type" class. Using this meta-model, we can derive instances, representing different parties and party types, and we can provide the respective constraints both on the instance-level and the meta-level.

Virtual Connector

- *Textual description*: In many patterns and larger architectures, components have no direct relationship, but still communicate virtually using other components and connectors in between. For instance, in a layered distributed client/server architecture a component on the client-side often virtually communicates with a component from the same layer on the server side.

- *Known uses in patterns*: BROKER [6], remoting patterns [38], remote PROXY [6].

- *Modeling issues*: The virtual relationship is an important additional information, but is not explicit in a UML diagram. It must be deduced from the implicit collaboration of components and connectors. If multiple virtual dependencies exist in the same architecture, as for instance in distributed layers, it cannot be deduced which component corresponds with which other component without further documentation.

- *Modeling solution*: We introduce a stereotype ≪*VirtualConnector*≫ as an extension of the Connector metaclass. This connector is used between components that have a virtual relationship. We further define the stereotype ≪*IVirtual*≫, as an extension of the Interface metaclass. Therefore a ≪*VirtualConnector*≫ matches an ≪*IVirtual*≫ Interface of one component to another. We enforce the constraint that the ≪*VirtualConnector*≫ can only be used between two components A and B, if there is a path of components and connectors that link A to B. For instance, if A is connected to C, C is connected to D, and D is connected to B, then a ≪*VirtualConnector*≫ from A to B might be introduced.

5. CASE STUDY

Leela [40] is an infrastructure that provides a federation of remote peers, thus offering loosely-coupled services. Within a federation, all peers are equal, they can offer Web services (and possibly other kinds of services) to other peers, and they can connect spontaneously to other peers (and to the federation). Each remote object can potentially be part of more than one federation as a peer, and each peer decides which services it provides to which federation. Certain peers in a federation may be able to access extra services that are not offered to other peers in this federation, via their partaking in other federations. Leela peers are hosted by Leela applications. One such application can host multiple peers and federations.

Leela's design follows the architectural patterns from [38]. In our first attempt to design the system, we used the standard UML class diagrams [40]. However, the architectural patterns could not be explicitly modeled and therefore the design decisions taken that were concerned with these patterns are not documented, except as complementary meta-information to the class diagrams. This meta-information can be textual or it can make use of a formal notation, nevertheless it is not part of the UML diagrams. To overcome this problem, we have applied our UML profile to explicitly model the architectural components, connectors, configurations, and constraints in Leela's design. Due to space constraints, as a case study we present an excerpt of this design: the basic communication framework of Leela.

5.1 Broker architecture

Leela implements a BROKER [6], which suggests a general architectural configuration that separates a distributed system's communication functionality from its application functionality by isolating all communication-related concerns. A BROKER hides and mediates all communications between the objects or components in a system. Local client-side and server-side brokers enable the exchange of requests and responses between the peers.

Each peer in Leela acts as a client and a server at the same time. Thus, Leela peers are composite components that contain both client-side and server-side BROKER sub-components. In the following description, the BROKER is viewed as a compound pattern that is implemented using several patterns from the Remoting pattern language [38] (the details are depicted in Figure 4). Even though client-side and server-side BROKER components are present in the same system, it makes sense to distinguish client-side and server-side roles of the components in order to make the pattern-based architecture more understandable. Unfortunately, this cannot be easily modeled with UML because the BROKER as a whole is not an explicit component, but consists of several components. Thus we cannot use UML composition or aggregation relationships here. However, the Grouping primitive from our UML profile is an ideal match. We introduce two ≪*Group*≫ packages: ClientBroker and ServerBroker. For each BROKER component, we add a namespace

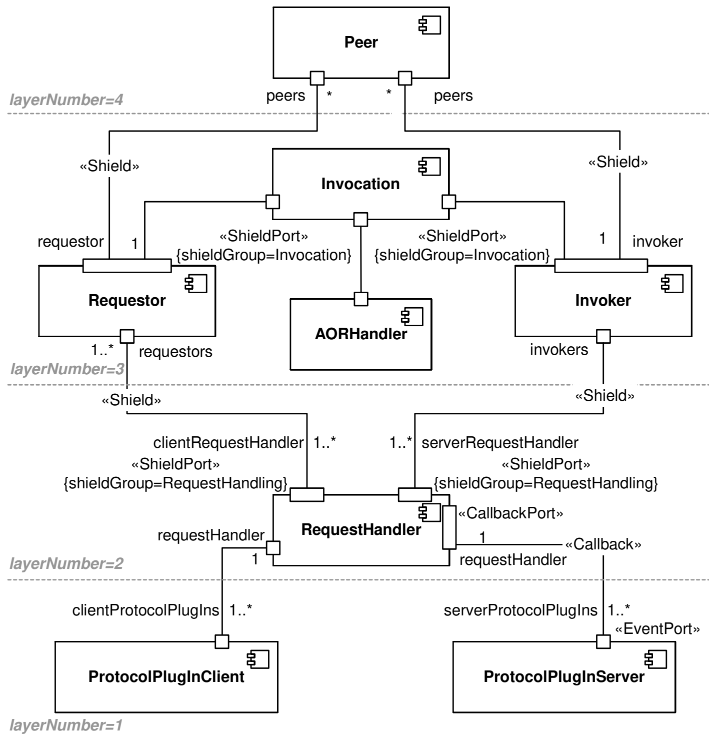

Figure 4: Basic broker-based invocation architecture

relationship either to the ClientBroker group or the ServerBroker group, indicating membership to the respective group. The group membership of the components introduced is depicted in Figure 5.

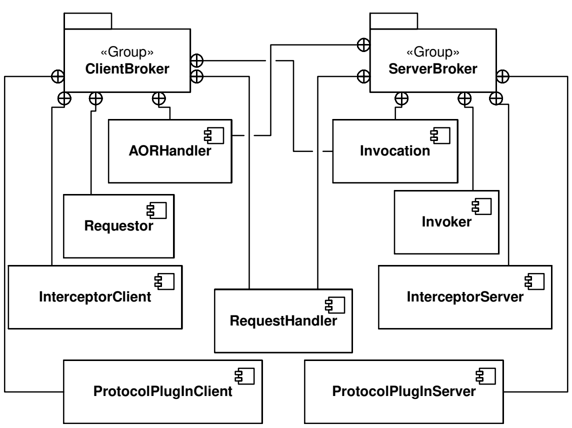

Figure 5: Group memberships of the Leela components

5.2 Basic invocation architecture

Figure 4 shows the basic software architecture diagram of Leela, using our profile. The BROKER consists of a client-side REQUESTOR to construct and forward invocations, and a server-side INVOKER [38] that calls the target peer's operations. A MARSHALLER is implemented by the Invocation and AORHandler components. These are used on each side of the communications path to handle the transformation of requests and responses from programming-language-native data types into byte arrays that can be sent over the wire. Thus they are member of both groups, ClientBroker and ServerBroker (see Figure 5).

As its basic communication resource each Leela application uses a component, called the RequestHandler, that implements both a CLIENT REQUEST HANDLER [38] and a SERVER REQUEST HANDLER [38]. The CLIENT REQUEST HANDLER forwards request messages from a client to the server. The SERVER REQUEST HANDLER receives these requests at the server side and triggers the invocation of the peer. Because RequestHandler realizes both patterns, it is member of both groups, ClientBroker and ServerBroker.

The request handlers contain PROTOCOL PLUG-INS [38] for the various protocols that transport the message across the network. Currently, Leela supports PROTOCOL PLUG-INS [38] for various SOAP implementations. However, virtually any other communication protocol can be used as well, because Leela's MARSHALLER [38] uses a simple string-based format as a message payload, and (re-)uses Tcl's automatic type converter to convert the string representations to native types and vice versa.

There are a number of further design issues which need to modeled. First of all, the application of the remoting patterns leads to an architecture based on the LAYERS pattern [6]. The same layers are present on client and server side: Protocol, RequestHandling, Invocation, and Application. We model the layers according to our Layering primitive. For each layer, we introduce a ≪*Layer*≫ package and the tagged value receives the respective layer number. Each layered component is imported to the corresponding layer. Figure 6 shows the layer membership of the components discussed in this section. There a number of constraints:

- Components from the layer Application can only interact with components from the layers Application and Invocation, or components who are not part of a layer.

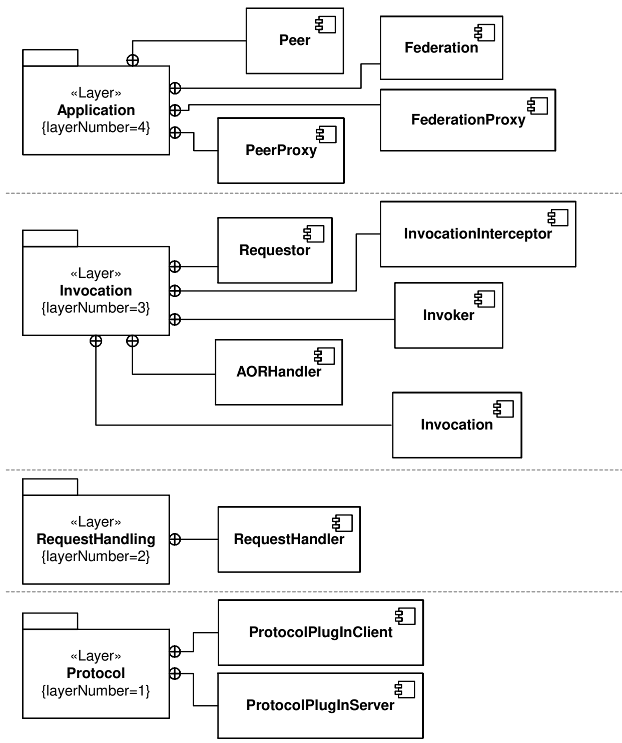

Figure 6: Layers of the Leela architecture

- Components in the layer Invocation can only be accessed via Invoker or Requestor, through a Shield Connector. That is, all internal interfaces are stereotyped ≪*IShield*≫.
- Components from the layer Invocation can only interact with components from the layers Invocation and RequestHandling, or components who are not part of a layer.
- Components in the layer RequestHandling can only be accessed via the RequestHandler component through a Shield Connector. That is, all internal interfaces are stereotyped ≪*IShield*≫.
- Components from the layer RequestHandling can only interact with components from the layers RequestHandling and Protocol, or components who are not part of a layer.
- Components from the layer Protocol can only interact with components from the layer Protocol or components who are not part of a layer.

Note that these constraints apply for client-side and server-side components, which, as aforementioned, are distinguished using the Grouping primitive. The client-side PROTOCOL PLUG-IN is simply invoked by the request handler component. The server-side PROTOCOL PLUG-IN, however, receives requests and result messages from the network asynchronously (it contains a REACTOR [33] implementation). Thus the request handler is informed of network events using callback events. This is modeled using our Callback primitive (see Figure 4).

In addition a virtual communication between the respective components at each layer of the BROKER architecture happens. This is modeled using the Virtual Connector primitive, as shown in Figure 7.

So far we have only modeled the base components. In the next sections, let us take a closer look at two exemplary component types: peers and interceptors.

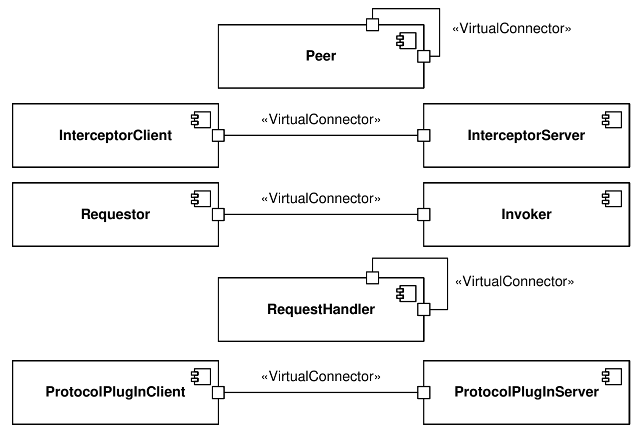

Figure 7: Virtual communication among Leela components

5.3 Peers and federations

As aforementioned, two different kinds of peers exist: ordinary peers and federations of peers. Federations, naturally, contain peers, but this cannot be properly modeled with UML's composition or aggregation relationship alone because we require a constrained relationship here. Thus we model federations as special, composite peers that are connected through an Aggregation Cascade to other peers with the following constraints:

- A peer can be part of multiple federations. That's why we use Aggregation Cascade and not Composition Cascade.
- A federation cannot contain peers of the type federation.

Peers can interact with other peers using the REQUESTOR, which realizes the virtual communication link. Sometimes it is more convenient to use the pattern CLIENT PROXY [38]: a CLIENT PROXY is a placeholder for the peer in the client process. By presenting clients with an interface that is the same as the peer's interface, the proxy lets the client interact with the peer as if it were a local object. Internally, the CLIENT PROXY transforms the invocations it receives into REQUESTOR invocations. Leela supports peer and federation proxies that act as CLIENT PROXIES, offering the interfaces of a peer or federation. The proxies thus provide indirections, which can be modeled using the Indirection primitive (see Figure 8).

5.4 Invocation interceptors

The Leela invocation chain on the client side and the server side is based on INVOCATION INTERCEPTORS [38], which transparently extend the invocation on both sides with new behavior. INVOCATION INTERCEPTORS are used for introducing many add-on tasks, such as logging and authentication. The most prominent task of the INVOCATION INTERCEPTORS in Leela is control of remote federation access. On the client side, an INVOCATION INTERCEPTOR intercepts the construction of the remote invocation and adds all federation information for a peer into the INVOCATION CONTEXT [38]. On the server side this information is read by another INVOCATION INTERCEPTOR. If the remote peer is not allowed to

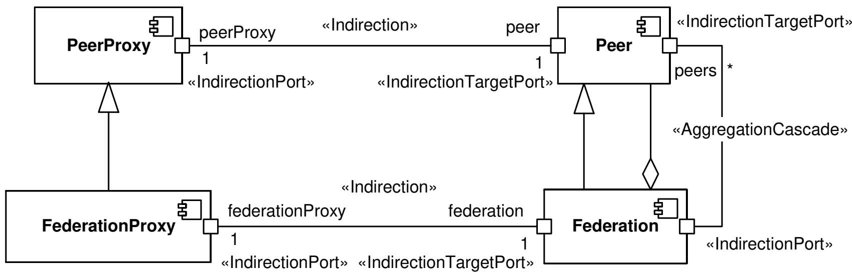

Figure 8: Proxy-based indirection in Leela

access the invoked peer, the INVOCATION INTERCEPTOR stops the invocation and sends a REMOTING ERROR to the client, otherwise access is granted. INVOCATION INTERCEPTORS are triggered by callbacks (modeled using the Callback primitive), as can be seen in Figure 9. Naturally the interceptors on the client-side and the server-side are linked through a Virtual Connector.

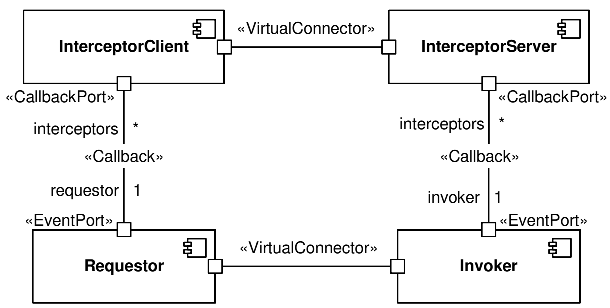

Figure 9: Invocation interceptors in the invocation chain

Often interceptors for one and the same task exist both on client-side and server-side. In Figure 10 three examples are presented. Logging is needed both on client-side and server-side, but no Virtual Connector between the logging interceptors is necessary. The server-side federation interceptor checks whether an invoking peer belongs to a federation or not. The client-side federation interceptors thus must put the federation information of the invoking peer into the INVOCATION CONTEXT. Thus there is a virtual communication between these two interceptors, which is modeled using a Virtual Connector. Likewise, the client-side and server-side authentication interceptors need to transmit authentication information over the wire.

6. RELATED WORK

The approach described in this paper is based on related research on architectural primitives, UML profiles for architectural description, and modeling architectural patterns.

The idea of primitives as the fundamental elements of architectural patterns and design patterns has been investigated from several viewpoints. Pree has worked in the area of object-oriented frameworks and has explored primitives in the construction of 'hot spots', i.e. variation points that are adapted in individual applications [30, 31]. His primitives are defined at two levels of abstraction: at a lower level, the fundamental elements of patterns are 'hook' and 'template' methods and their corresponding classes; at a higher level the aforementioned fundamental elements are used to specify composition patterns for hot spots that are called *meta-patterns*. These composition patterns themselves can be used for specifying even higher-level patterns, such as the GoF [11] patterns; however they are not architectural elements and thus cannot be used to describe architectural patterns like the architectural primitives in this paper.

In the realm of patterns, many patterns are described as compound patterns that consist of other, existing patterns. For instance, in [38] the BROKER pattern is described as a compound pattern composed from patterns from [38, 33, 11, 6]. Our approach follows a similar philosophy as we define primitives that can be used to model architectural patterns, but is different in that these architectural primitives are more specific and formally specified than patterns. The primitives can be seen as participants of patterns, whereas patterns require substantial hand-crafting (i.e. a design and implementation effort) in order to be used as part of another pattern.

Mehta and Medvidovic proposed a framework for composing architectural patterns through *architectural primitives* [24] that are certain underlying concepts, shared by all patterns. They propose a number of such primitives as the building blocks for constructing the architectural elements of patterns and demonstrate their approach through the representation of several architectural patterns through the primitives. This approach is based on the assumption that there exists a fixed set of fundamental primitives that can potentially characterize any architectural pattern participant and therefore this framework of primitives can be used for characterizing and comparing patterns. Our approach is different in the sense that we investigate architectural primitives at a larger granularity and a higher level of abstraction. Moreover, our primitives are recurring concepts in several, but not all, architectural patterns, and they are characterized by rich semantics that serve specialized purposes. Similarly, Bass et al. in [3] have also proposed a predefined set of *unit operations*, such as separation, abstraction, compression and resource sharing as the building blocks for all architectural and design patterns. In contrast to our architectural primitives, these unit operations are defined at a much higher level of abstraction. They rather describe atomic architectural transformations and operations, whereas our primitives describe fundamental, recurring structures.

There have also been several attempts for specifying existing ADLs or proposing new ADLs as extensions of UML, usually in the form of profiles. Medvidovic et al. have pointed out three different ways to use UML as an ADL [22]: (a) using the "pure" UML metamodel "as is", which forces the architect to implicitly define the necessary architectural concepts; (b) constraining the UML metamodel through profiles and thus providing explicitly the

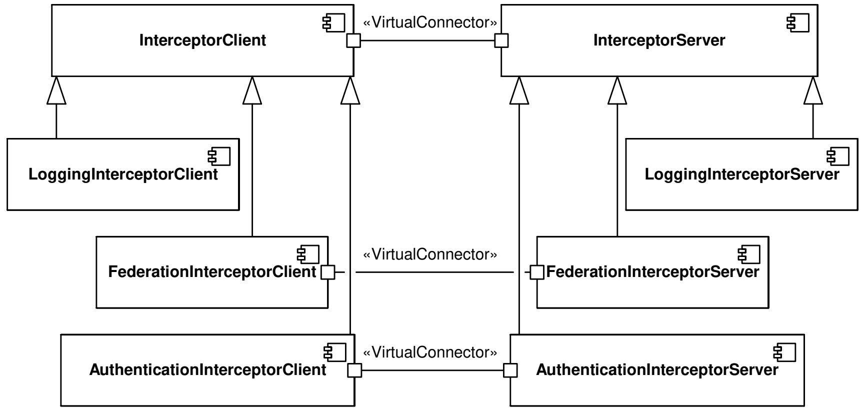

Figure 10: Special invocation interceptors

architectural concepts as constrained stereotypes, while still conforming to the standard metamodel; (c) modifying the UML metamodel and thus providing "native" support for architectural description, but losing conformance to the standard metamodel. They have also evaluated the first two approaches by using them to map three ADLs to UML. Clements et al. in [7] demonstrated how UML 1.x can be used "as is" in representing the fundamental architectural concepts in a number of architectural views. This work was continued by Garlan et al. in [12] and later by Ivers et al. in [17] to take under account the forthcoming UML 2.0, and particularly focus on the provision for the component and connector view in the new standard. The improvements of the new UML 2.0 metamodel for architectural concepts, especially ports and internal structures, was also advocated by Björkander and Kobryn in [4]. Finally Selic and Rumbaugh [34, 35] have defined a UML profile for real-time systems, UML-RT, which embodies several architectural concepts such as components (so-called "capsules"), connectors, and ports. Our approach uses a different line of attack: we do not model the architectural concepts that are specific to an architectural pattern, but rather the fundamental primitives that participate in a number of patterns. Thus we overcome the limitations of ADLs by providing a wealth of abstractions, capable of modeling several of the well-known architectural patterns.

There are many approaches for modeling or representing software patterns, the vast majority of which focuses on the design patterns from [11]. A number of such approaches attempted to formally specify these patterns (see for instance [8, 25, 37, 21]). These approaches, however, have not gained much momentum, mainly because of their complexity and the resulting limitations regarding their practical use. These approaches have not been used for architectural patterns or whole pattern languages, like our primitives, but just for some isolated patterns from [11]. A third major difference of these approaches, compared to our approach, is that they only support one variant of a pattern (often simply following the C++ example from [11]) and not other possible pattern variations. The same problem appears also when using the Collaboration metaclass provided by UML 2.0 to describe a design pattern. Most patterns (especially architectural patterns), however, can be realized using a multitude of different design variants. Our approach describes primitives that are participants of the patterns and can be tailored to support multiple variants of a pattern. In other words, we can model the variants of a pattern, by constraining the specific semantics of the architectural primitives that comprise the pattern.

There have also been some approaches that propose language support for design patterns, such as [27, 5], or implementations of patterns as aspects, such as [13, 14]. These approaches make patterns first-class entities of the language or aspect framework, and thus they become more traceable in the code than a pattern implementation scattered across a number of classes. All of these approaches might be considered as a way to better understand the use of a single pattern in an architecture, but not for documenting the design of complex architectures based on (multiple) patterns, as this paper advocates.

7. CONCLUSIONS AND FUTURE WORK

We have proposed modeling architectural patterns through a number of architectural primitives in the component and connector view. We have elicited an initial set of these primitives from a pool of established architectural patterns in order to ensure their correctness and broad applicability. This set of primitives is original and helps solving the fundamental problems in modeling architectural patterns that were outlined in Section 1: they offer the necessary abstractions that grasp the rich semantics found in patterns; they can represent not only a specific pattern variant but multiple variants of a pattern, by tailoring the architectural primitives with constraints.

We have selected UML, a de-facto standard modeling language in software architecture, in order to guarantee tool support and familiarity of modelers. However the the main shortcoming of this approach stems from the very own use of UML. The extension mechanisms of UML, in particular the stereotypes, are awkward to use because of their second-class status: they are neither metaclasses of the standard metamodel, nor model elements and this fact often confuses the users of UML. Furthermore, even though OCL constraints provide semi-formal semantics to the stereotypes, they are not well accepted in the software architecture community, partly because there are no tools so far that can dynamically check the constraints in UML models. Lastly, the constant evolution of the UML standard, forces us to update the mapping of the architectural primitives in the language in its subsequent versions, which can prove to be cumbersome. However, we do believe that the advantages that UML conveys outweigh these disadvantages.

We plan to extend this work in the following directions:

- document the architectural primitives of other domain-specific patterns and pattern languages in the component and connector view;

- further work on the relation of our approach to the notion of pattern languages (larger collections of interrelated patterns). In particular, we plan to document more patterns from the remoting pattern language (see [38]) and a pattern language for general-purpose architectural patterns (see [2]);
- search for architectural primitives in other views, such as the module view.

8. REFERENCES

[1] P. Avgeriou, N. Medvidovic, and N. Guelfi. Software Architecture Description with UML. In J. Nunes, B. Selic, A. Silva, and A. Toval, editors, *UML Modeling Languages and Applications - UML 2004 Satellite Activities*, Lisbon, Portugal, October 2004. Springer Verlag, Volume 3297 of LNCS.

[2] P. Avgeriou and U. Zdun. Architectural patterns revisited – a pattern language. In *10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, Irsee, Germany, July 2005.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice 2nd Edition*. Addison Wesley, Reading, MA, USA, 2003.

[4] M. Björkander and C. Kobryn. Architecting systems with UML 2.0. *IEEE Softw.*, 20(4):57–61, 2003.

[5] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[8] A. H. Eden and Y. Hirshfeld. LePUS – symbolic logic modeling of object oriented architectures: A case study. In *Second Nordic Workshop on Software Architecture - NOSA'99*, Ronneby, Sweden, April 1999.

[9] T. Foster and L. Zhao. Cascade. *Journal of Object-Oriented Programming*, 11(9), Feb. 1999.

[10] M. Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] D. Garlan, S.-W. Cheng, and A. J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49, 2002.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, Nov. 2002. ACM Press.

[14] R. Hirschfeld, R. Lämmel, and M. Wagner. Design Patterns and Aspects — Modular Designs with Seamless Run-Time Integration. In *Proc. of the 3rd German GI Workshop on Aspect-Oriented Software Development, Technical Report, University of Essen*, Mar. 2003. 8 pages.

[15] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[16] IEEE. Recommended Practice for Architectural Description of Software Intensive Systems. Technical Report IEEE-std-1471-2000, IEEE, 2000.

[17] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva. Documenting component and connector views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, 2004.

[18] R. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[20] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[21] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in UML. In *Proceedings of the 26th International Conference on Software Engineering*, pages 252–261. IEEE Computer Society, 2004.

[22] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[23] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[24] 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*, pages 347–350, Helsinki, Finland, 2003. ACM Press.

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

[26] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, April 1996.

[27] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.

[28] OMG. UML 2.0 superstructure final adopted specification. Technical Report ptc/03-08-02, Object Management Group, August 2003.

[29] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[30] W. Pree. Metapatterns: A Means for Capturing the Essentials of Object-Oriented Design. In *European Conference on Object-Oriented Programming, (ECOOP), Bologna, 4-8 July 1994*. Springer-Verlag.

[31] W. Pree. Hot-spot-driven framework development. In R. J. M. Fayad, D. Schmidt, editor, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley & Sons, 2000.

[32] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S.

Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th ICSE*, pages 209–218, Kyoto, Japan, 1998. IEEE Computer Society.

[33] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[34] B. Selic. Turning clockwise: using UML in the real-time domain. *Commun. ACM*, 42(10):46–54, 1999.

[35] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. 1998.

[36] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.

[37] 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.

[38] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns*. Pattern Series. John Wiley and Sons, 2004.

[39] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[40] U. Zdun. Loosely coupled web services in remote object federations. In *Proceedings of the Fourth International Conference on Web Engineering (ICWE'04)*, Munich, Germany, July 2004.

[41] U. Zdun. Some patterns of component and language integration. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPlop 2004)*, Irsee, Germany, July 2004.