In *Proceedings of the Third International Conference on the Unified Modeling Language – << UML >> 2000,*
October, 2000, York, UK.

Reconciling the Needs of Architectural Description with Object-Modeling Notations

David Garlan and Andrew J. Kompanek

Carnegie Mellon University
Pittsburgh, PA 15213 USA
{garlan, kompanek}@cs.cmu.edu

Abstract. Complex software systems require expressive notations for representing their software architectures. Two competing paths have emerged. One is to use a specialized notation for architecture – or architecture description language (ADL). The other is to adapt a general-purpose modeling notation, such as UML. The latter has a number of benefits, including familiarity to developers, close mapping to implementations, and commercial tool support. However, it remains an open question as to how best to use object-oriented notations for architectural description, and, indeed, whether they are sufficiently expressive, as currently defined. In this paper we take a systematic look at these questions, examining the space of possible mappings from ADLs into object notations. Specifically, we describe (a) the principle strategies for representing architectural structure in UML; (b) the benefits and limitations of each strategy; and (c) aspects of architectural description that are intrinsically difficult to model in UML using the strategies.

1 Introduction

A critical level of abstraction in the description of a complex system is its software architecture. At an architectural level one describes the principal system components and their pathways of interaction. Architectural descriptions are typically used to provide an intellectually tractable, birds-eye view of a system, and to permit design-time reasoning about system-level concerns, such as performance, reliability, portability, and conformance to external standards and architectural styles.

In practice most architectural descriptions are informal documents. They are usually centered on box-and-line diagrams, with explanatory prose. Visual conventions are idiosyncratic, and usually project-specific. As a result, architectural descriptions are only vaguely understood by developers, they cannot be analyzed for consistency or completeness, they are only hypothetically related to implementations, their properties cannot be enforced as a system evolves, and they cannot be supported by tools to help software architects with their tasks.

To improve the situation a number of people have suggested the use of more standardized and formal notations for architectural description. Viewed broadly, there are two main sources of such recommendations. One is from the software architecture research community, which has proposed a number of "architecture description languages" (ADLs) specifically designed to represent software and system architectures [17]. These languages have matured over the past five years. Most come with tool sets that support many aspects of architectural design and analysis, such as graphical editing, code generation, run-time monitoring, anomaly detection, and performance analysis.

The other source is from the object-oriented community. Object modeling notations have had considerable success, both as domain-modeling and implementation-modeling notations. For the former, object-oriented entity-relation diagrams (and related notations) have provided a natural mechanism to represent domain entities and their relations. In the latter case, for systems that are to be implemented in an object-oriented fashion, object-modeling notations provide a natural way to represent class structures, relationships between those classes, and certain aspects of class/system behavior.

But are object-oriented notations also appropriate as architectural descriptions? A number of authors have argued yes, claiming that the current capabilities are precisely what is needed [3, 13]. Moreover, recently there have been proposals that attempt to show how ADL concepts can be mapped directly into a notation like UML [11,16,18].

Unfortunately, each proposal is different, and each covers only *some* aspects of architectural description. We believe that what is needed is more systematic view of the problem, one that clarifies what needs to be modeled at an architectural level, identifies candidate constructs in an object modeling language, and examines the strengths and weaknesses of adopting a particular modeling strategy. Practitioners could then decide more rationally which, if any, of the possible techniques is most appropriate.

In this paper we attempt to do that for the *structural* aspects of software architecture. We start by examining the needs for architectural description—based on what is now done informally in practice and what features are captured by ADLs. Then we consider a number of strategies for representing architectures in UML, contrasting the advantages and drawbacks of each. Finally, we summarize the key lessons from this activity.

2 Related work

A number of authors have examined ways to model architectures using object notations. Among the earliest of these were purveyors of object-oriented methods, who attempted to provide a uniform path from requirements to implementations using only object notations. In these treatments architecture is usually not specifically called out as a representation in need of special treatment, but viewed as a form of high-level object-oriented design.

The lack of explicit treatment of architectures in those methods prompted several authors to take a more careful look at the needs of architectural modeling. Kruchten [13] for example, proposes a set of (4+1) views to capture the various aspects of architectural information. Other authors have examined the use of object-oriented notations for modeling architectural styles and patterns [3]. Finally, profiles have been proposed within the UML standards community for commercial modeling languages that include explicit architectural concepts, including a profile for Corba [23] and real-time systems [28].

Recently, there have been a number of attempts to understand more broadly how to *map* architectural models as expressed in an architecture description language into object notations [11,16,18,19]. As we noted earlier, each of these proposes a particular form of modeling that exploits the constraints of the particular architectural domain or language. Unlike this paper, they have not attempted to consider systematically the *space* of possible embeddings, or enumerate the situations under which *alternative* embeddings might be desirable – although they do serve as relevant data points. On the other hand, for specific mapping strategies many of these efforts go beyond our paper by also considering mappings of non-structural aspects to UML, such as architectural behavior.

3 Architectural Description

The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. Currently there is considerable diversity in the ways that practitioners represent architectures, although most depend on informal box-and-line diagrams.

In an effort to put architectural description on a more solid notational and semantic footing, over the past decade a number of architecture description languages have been proposed [1,2,6,9,14,15,18,20,29]. While these languages (and their associated tool sets) differ in many details, there has emerged a general consensus about the main ingredients of architectural description. Focusing on architectural structure, we take that core set of concepts as the starting point for this paper[1]. In this shared ontology there are five basic concepts: components, connectors, systems, properties, and styles.

Components represent the computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include clients, servers, filters, blackboards, and databases. Components may have multiple interfaces (which we will call *ports*), each interface defining a point of interaction between a component and its environment. A component may have several ports of the same type (e.g., a server may have several active http connections).

Connectors represent interactions among components. They provide the "glue" for architectural designs, and correspond to the lines in box-and-line descriptions. From a run-time perspective, connectors mediate the communication and coordination activities among components. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. Connectors may also represent complex interactions, such as a client-server protocol or a SQL link between a database and an application. Connectors have interfaces that define the *roles* played by the participants in the interaction.

Systems represent graphs of components and connectors. In general, systems may be hierarchical: components and connectors may represent subsystems that have *internal* architectures. We will refer to these as *representations.* When a system or part of a system has a representation, it is also necessary to explain the mapping between the internal and external interfaces. We will refer to the elements of this mapping as *bindings.*

Properties represent additional information (beyond structure) about the parts of an architectural description. Although the properties that can be expressed by different ADLs vary considerably, typically they are used to represent anticipated or required extra-functional aspects of an architectural design. For example, some ADLs allow one to calculate system throughput and latency based on performance estimates of the constituent components and connectors [30]. In general, it is desirable to be able to associate properties with any architectural element in a description (components, connectors, systems, and their interfaces). For example, an interface (port or role) may describe an interaction protocol.

Types and Styles represent families of related systems. An architectural *style* typically defines a vocabulary of design element types as a set of component, connector, port, role, binding, and property types, together with rules for composing instances of the types [10].

[1] These concepts are largely those in found in Acme [7].

Examples include data-flow architectures based on graphs of pipes and filters, blackboard architectures, and layered systems.

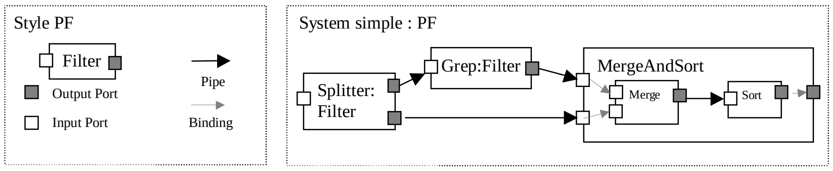

Fig. 1. A system in the Pipes and Filters style

To illustrate the use of these concepts, consider the simple example shown in Figure 1, which we will use throughout the paper. The system defines a simple string-processing application that extracts and sorts text. The system is described in a pipe-filter style, which provides a design vocabulary consisting of a filter component type and pipe connector type, input and output port types, and a port-port binding type. In addition, there would likely be constraints (not shown) that ensure, for example, that the reader/writer roles of the pipe are associated with appropriate (input/output) ports. The system is described hierarchically: *MergeAndSort* is defined by a representation that is itself a pipe-filter system. Properties of the components and connectors (not illustrated) might list, for example, performance characteristics used by a tool to calculate overall system throughput.

4 UML Background

UML unifies a number of object modeling notations in a common framework and is quickly becoming a standard object notation for industry [4,5,26,27]. While a detailed description of UML is beyond the scope of this paper, we summarize its principal constructs (known as model *elements*) that can be used to model software architectures:

Classes, Interfaces and Objects: Classes are the primary construct for describing the logical view of a system. Classes have properties in the form of *attributes*, provide abstract services in the form of *operations*, and can be logically related to one another using *associations*. Classes may expose their functionality through a set of supported *interfaces*, collections of related operations. Classes have instances called *objects*, which are used in models called *collaborations* to depict behavior under particular scenarios.[2]

Component and Component Instances: Components are used to describe the physical, deployable pieces of a system. Like classes, components in UML expose their functionality through interfaces. Components are typically related to each other using dependency relationships. The deployment of a system on a set of hardware is described by associating components with hardware *nodes*.

Packages: UML provides a grouping mechanism that is used to partition large UML models into manageable chunks called *packages*. UML also defines a type of grouping element called a *subsystem,* which is typically used to encapsulate the object models that define a coarse-grained module in a system.

Relationships: Model elements are related to one another with *associations* and *dependencies*. Dependency is the most generic relationship in UML, indicating that an element depends in some way on the definition of another element. Association is a richer

[2] In the UML meta-model, the prototypical "objects" in collaborations are referred to as collaboration *roles*.

relationship that describes an abstract relationship between classes and the roles the classes play in the relationship.

Stereotypes: To allow the extension of UML with domain-specific concepts, UML provides a mechanism for associating constraints with elements of a model, using a constraint language, *OCL* [25]. These constraints can be grouped and named using a construct called a *stereotype*. UML also includes a set of standard stereotypes.

The above constructs can be composed in various ways in a UML model and visualized in diagrams. Textual *annotations* may be associated with any of them. Frequently, these annotations are in the form of *tagged values:* arbitrary attribute-value pairs.

UML also defines a set of models for describing the dynamic behavior of a system, including collaboration diagrams that specify system behavior using event-based *interaction scenarios*, descriptions based on *state machine*s, and *use cases*.

5 Strategies & Evaluation Criteria

We now consider five strategies for modeling architectural structure (as expressible in ADLs) using UML. We will organize the presentations of alternatives around the choices for representing component types and instances, since the components are typically the central design elements of an architectural description. For each choice we then consider sub-alternatives for the other architectural elements (connectors, styles, etc.). Of the five strategies, the first three consider ways to use classes and objects to model components. The fourth is based on UML components, and the fifth on UML subsystems.

However, before considering those strategies, we need to be clear about the criteria for evaluation. Ideally we would like a mapping strategy to exhibit three characteristics:

1. *Semantic Match:* It should respect documented UML semantics and the intuitions of UML modelers. The interpretation of UML model should be close to the interpretation of the original ADL description so the model is intelligible to both designers and UML-based tools. In addition, the mapping should produce legal UML models.
2. *Legibility:* The resulting architectural descriptions in UML should bring conceptual clarity to a system design, avoid visual clutter, and highlight key design details.
3. *Completeness:* The architectural concepts identified in Section 3 should be representable in the UML model.

6 Classes & Objects – Types as Classes, Instances as Objects

Perhaps the most natural candidate for representing component and connector types in UML is the class concept. Classes describe the conceptual vocabulary of a system just as component and connector types form the conceptual vocabulary of an architectural description in a particular style. Additionally, the relation between classes and instances is similar to the relationship between architectural types and their instances. The mapping described for the C2 architectural style in [16] is a variation on this approach.

Figure 2 illustrates the general idea. Here we characterize the filter architectural type as the UML class "filter." Instances of filters, such as "split" are represented as corresponding objects in an object (instance) diagram. We now take a closer look at this strategy by examining how the concepts in Section 3 can be described in UML.

6.1 Components

As noted, the type/instance relationship in architectural descriptions is a close match to the class/object relationship in a UML model. In UML, classes, like component types in architectural descriptions, are first-class entities and are the richest structures available for capturing software abstractions. The full set of UML descriptive mechanisms are

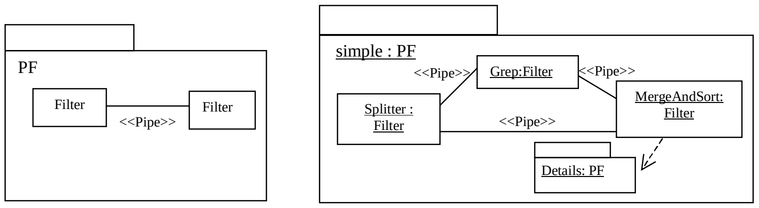

Fig. 2. Types as classes, instances as objects.

available to describe the structure, properties, and behavior of a class, making this a good choice for depicting detail and doing analysis using UML-based analysis tools.

Properties of architectural components can be represented as class attributes or with associations; behavior can be described using UML behavioral models; and generalization can be used to relate a set of component types. The semantics of an instance or type can also be elaborated by attaching one of the standard stereotypes (e.g., indicating that a component runs as a separate process with the <<process>> stereotype).

However, the match between component types and classes, and component instances and objects, is not perfect. In architectural descriptions a component instance often has additional structure not required by the component's type. For example, a component instance might define additional ports not required by its type, or associate an implementation in the form of additional structure that is not part of its type's definition. However, in UML an object cannot include parts that its class does not also define.

6.2 Ports (Component Interfaces)

There are six reasonable ways to represent ports.

Option 1: No explicit representation. We can leave ports out of the model entirely. This leads to the simplest diagrams, but suffers from the obvious problem that there is no way to characterize the names or properties of the ports. However, this might be a reasonable choice in certain situations, such as when components have only a single port, or when the ports can be inferred from the system topology.

Option 2: Ports as Annotations. We can represent ports as annotations. This approach provides a home for information about ports, although annotations have no semantic value in UML and hence cannot be used as a basis for analysis. Again, if the detailed properties of a port are not of concern this might be a reasonable approach.

Option 3: Ports as Class/Object Attributes. Ports can be treated as attributes of a class/object. In this approach ports are part of the formal structural model, but they can have only a very simple representation in a class diagram – essentially a name and type.

Option 4: Ports as Interfaces. Describing port types as UML interfaces has three advantages. First, the interface and port concepts have a similar intent: they both characterize aspects of the ways in which an entity can interact with its environment. Second, the UML "lollipop" notation provides a compact description of a port in a class diagram depicting a component type. In an instance diagram, a UML association role (corresponding to a port instance) qualified by the interface name (the port type) provides a compact way to designate that a component instance is interacting through a particular port instance. Finally, this approach provides visually distinct depictions of components and ports, in which ports can clearly be seen as subservient to components.

However, while the interface and port concepts are similar, they are not identical. An interface exposes a set of operations that can be invoked by the environment of a component. In contrast, the description of a port in an ADL often includes both the services *provided* by the component, as well as those it *requires* from its environment. Furthermore, it is meaningful for a component type to have several instances of the same port type, while it is not meaningful to say that a class realizes several versions of the same. For example, there is no easy way to define a "splitter" filter type that has two output ports of the same "type" using this technique. Finally, unlike classes, interfaces do not have attributes or substructure.

Option 5: Ports as Classes. Another alternative is to describe ports as classes contained by a component type. This is essentially the approach taken in the UML Real-Time Profile [22,28]. This overcomes a certain lack of expressiveness in UML interface descriptions: we can now represent port substructure and indicate that a component type has several ports of the same type. A component instance is modeled as an object containing a set of port objects. Unfortunately, this approach also suffers from problems: by representing ports as classes, we not only clutter the diagram, but we also lose clear discrimination between ports and components. It is possible to make the diagram more suggestive using a notational variation on this scheme in which the ports are contained classes. But then indicating points of interaction is counterintuitive, as containment usually indicates that a class owns other classes whose instances may or *may not* be accessible through instances of the parent class.

Option 6: Ports as UML Classes that realize interfaces. The final option is a combination of options 4 and 5: represent ports as classes which themselves expose interfaces. This option is more expressive than the previous two techniques, but suffers from the semantic mismatch problem of both options. It is also the least visually appealing. Unless ports are discriminated from components visually, the added clutter in a diagram would mask the overall topology of components, defeating one of the main purposes of architectural description. It also makes interpretation more difficult because a reader is expected to understand that a pair of objects stands for a single object in the original model.

6.3 Connectors

Option 1: Types as Associations; Connectors as Links. In an architectural box-and-line diagram of a system, the lines between components are connectors. One tempting way to represent connectors in UML is as associations between classes or links between objects. The approach is visually simple, provides a clear distinction between components and connectors, and makes use of the most familiar relationship in UML class diagrams: association. Moreover, associations can be labeled, and when a direction is associated with the connector it can be indicated with an arrow in UML.

Unfortunately, although the identification between connectors and associations is visually appealing, connectors have a different meaning than associations. A system in an architectural description is built-up by choosing components with behavior exposed through their ports and connecting them with connectors that coordinate their behaviors. A system's behavior is defined as the collective behavior of a set of components whose interaction is defined and limited by the connections between them. In contrast, while an association or link in UML represents a potential for interaction between the elements it relates, the association mechanism is primarily a way of describing a conceptual relationship between two concepts.

Using an association to represent architectural connection has other problems. Since associations are relationships *between* UML elements, an association cannot stand on its own in a UML model. Consequently, there is no way to represent a connector type in isolation. To have any notion of a connector type within this scheme, one must resort to naming conventions or the use of stereotypes whose meaning is captured by an OCL description. Also, the approach does not allow one to specify the interfaces to the connector (i.e., its roles).

Option 2: Types as Association Classes. One solution to the lack of expressiveness is to qualify the association with a class that represents the connector type. This way the attributes of a connector type or connector can be captured as attributes of a class or object. Unfortunately, this technique still does not provide any way of explicitly representing connector roles. The approach is similar to the one taken in the UML Real-time Profile, in which association endpoints are identified with roles in a collaboration [28].

Option 3: Types as Classes; Connectors as Objects. One way to define roles in UML is to represent a connector as a class that is associated with model elements representing roles. We have the same options for representing roles as we had for ports: as interfaces realized by a class, as "child" classes contained by a connector class or as child classes that realize interfaces. Given a scheme for representing ports and roles, an attachment (between a port and a role) may be represented as an association or a dependency.

6.4 Systems

Option 1: Systems as UML Subsystems. The primary mechanism in UML for grouping related elements is the package. In fact, UML defines a standard package stereotype, called <<subsystem>>, to group UML models that represent a logical part of a system. The choice of subsystems is appropriate for *any* choice of mappings of components and connectors and works particularly well for grouping classes.

Unfortunately, subsystems have different semantics than systems in an architectural description. In a model, a package represents set of elements that may be imported into another context, but not a structure per se. In contrast, a system in architectural design is a structure with sub-parts in the form of its components and connectors. Unlike classes, packages also lack attributes for defining system level properties.

Option 2: Systems as Objects. A second option is to use objects to represent systems. If architectural instances are represented as objects, we can introduce an explicit system class whose instances contain the component and connector objects that make up the system. Then we can capture richer semantics using attributes and associations/links. Unfortunately, this approach has the drawback that by representing a system in the same way as a component or connector we lose the semantic distinction between a system as a configuration of elements. It is also introduces visual clutter.

Option 3: Systems as Subsystems Containing a System Object. This alternative is a combination of the first two techniques. It combines the expressiveness of Option 2 with the visual advantages of Option 1. However, we still have the basic semantic mismatch and the additional clutter complicates a diagram.

Option 4: Systems as Collaborations. A set of communicating objects (connected by links) is described in UML using a collaboration. If we represent components as objects, we can use collaborations to represent systems. While this is a natural way to describe run-time structures, unfortunately this leaves no way to explicitly represent system-level properties. There is also a semantic mismatch – a collaboration describes a representative interaction between objects that provides a partial description, whereas an architectural configuration is meant to capture a complete description at some level of abstraction.

There are of course many variations on Options 2 and 3, corresponding to the same variations on the use of classes, objects and stereotypes to describe components which we described in the last section and in the next.

6.5 Representations

A representation is just a system that is related to a component or connector and includes bindings that describe the relationship between the element and (one of) its representations. If systems are represented as packages, an element can be related to its representations through dependency, possibly with an annotation indicating the name of the representation. There are a number of options for representing bindings: (1) as annotations (a weak depiction); (2) as dependency between an outer element and inner element (only possible with techniques in which ports and roles are explicitly represented); (3) as set of elements with attributes that describe binding (visually cluttered). Alternatively, if systems are represented as classes, representations can be associated with their parent element using associations and links.

7 Classes & Objects – Types as Stereotypes, Instances as Classes

The second major alternative for modeling a component type in UML is to define a stereotype. In this way, we can describe the meaning of our architectural vocabulary in a way that distinguishes an architectural element type from the UML class concept. A component instance is then represented as a class with a stereotype. Using this approach, architectural concepts become distinct from the built-in UML concepts, and in principal, a UML-based modeling environment can be extended to support the visualization and analysis of new architectural types within a style and enforce design constraints captured in OCL. This is essentially the approach taken in [18].

In Figure 3, the Filter Type is defined by a set of constraints expressed in OCL that are identified with the <<Filter>> Stereotype. Filter instances (e.g., splitter) are represented as classes that bear the Filter stereotype. We now examine this approach in more detail.

7.1 Components

Representing instances as classes has a number of advantages. For example, we can capture a set of possible run-time configurations (e.g., one-to-many relationships) in a single diagram. By using classes, we also allow component instances to include structure in addition to the structure defined by their types, overcoming a limitation of the class/instance approach.

Unfortunately, the approach has a number of disadvantages. Stereotypes are not first class, so we can't define structure, attributes, or behavior except by writing OCL constraints. Furthermore, there is no way to visualize stereotypes in diagrams, unless, in the future, there is support in UML environments for manipulating, visualizing and analyzing extensions to the UML meta-model. There is also currently no way to express sub-typing relationships between stereotypes. Among other consequences, this means that we can't take advantage of analysis capabilities offered by UML tools to analyze architectural types, or associate behavioral models with types.

Furthermore, a class may have only one stereotype. Consequently, we can't use any the other built-in stereotypes to add meaning to a component that already has a stereotype. Arguably, using a *class* to represent an architectural *instance* is also not intuitive.

There are a number of options for representing ports. The ports defined by a component type would be represented as OCL expressions that state what structure a class standing for a component of this type should have. We can represent a component instance's ports in the same ways we modeled a component type's ports in last approach.

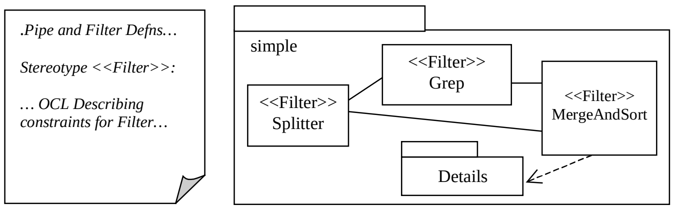

Fig. 3. Types as stereotypes, instances as stereotyped classes.

7.2 Connectors, Systems, Representations, and Styles

The options for representing connectors are similar to the options we had for representing connector types in the last approach. The same options exist for describing overall systems and for indicating representations as for the first variation, although in this we represent system instances as represented styles in the previous approach. In this case, there can be no explicit representation of a style using UML model elements. Instead, the style is embodied in a set of stereotypes.

8 Classes & Objects – Types as classes, Instances as Classes

The third class-based approach is to represent component types as classes (like the first approach) and component instances as classes (like the second). The observations we made in the previous two sections about the suitability of the classes to represent component types and instances apply here as well: by representing both component types and instances as classes, we have the full set of UML features to describe both component types and instances. We can also capture patterns at both the type (as part of a description of an architectural style) and instance level, supporting the description of a dynamic architecture whose structure evolves at run-time.

Figure 4 illustrates this approach, defining both the "filter" type and instances of filters (e.g., "splitter") as classes. We now examine this approach in light of the previous two class-based approaches.

8.1 Components

Although, the approach has many of the strengths of the previous two approaches, it suffers from a number of problems. Unlike the second approach, we no longer explicitly distinguish architectural vocabulary and the UML class concept. Representing both types and instances as classes also blurs the distinction between type and instance, although stereotypes may be used to reinforce this distinction.

There are two viable options for relating a component type to its instances within this scheme: generalization and dependency. Generalization captures the structural relationship between a type and instance (namely, the instance "subclass" must be substitutable for the type) but it blurs the type-instance distinction. The other option is to represent this

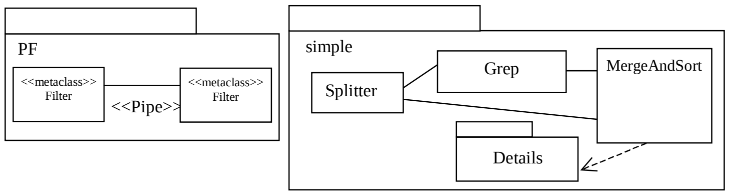

Fig. 4. Types & instances as classes.

relationship as a (perhaps stereotyped) dependency, which is semantically weaker but carries less semantic baggage than the generalization relationship.

8.2 Connectors

The same issues arise when describing connections. For a full discussion of options for describing connections in terms of classes, see variation #1.

8.3 Systems, Representations and Styles

The options for representing systems and styles are similar to those in the two previous approaches using (a) packages, or (b) classes and objects.

9 Using UML Components

UML includes a "component" modeling element, which is used to describe *implementation* artifacts of a system and their deployment. A component diagram is often used to depict the topology of a system at a high level of granularity and plays a similar function, although at the implementation level, as an architectural description of a system. In Figure 5 we represent the "filter" type as a UML component and instances (e.g., "splitter") as instances of this UML component.

9.1 Components

UML components are a natural candidate for representing architectural components. Components have interfaces, may be deployed on hardware, and commonly carry a stereotype and are depicted with a custom visualization. UML components are often used as part of diagrams that depict an overall topology, and just as it is natural to talk about mapping architectural components to hardware, components are assigned to nodes in UML deployment diagrams. For some architectural styles, the identification of abstract components with implementation-level components is a reasonable choice.

Unfortunately, in UML components are defined as concrete "chunks" of implementation (e.g., executables, or dynamic link libraries) that realize abstract interfaces – unlike the more abstract notion of components found in ADLs, which frequently have only an indirect relationship to deployable piece of a system. Nonetheless, the concepts share more than a name. Components expose interfaces (as with classes) and can be used to represent the ports exposed by a component, just as they were used in the strategy based on classes and objects.

However, the rich set of class associations available to relate classes are not available for components, limiting how we can describe ports, represent patterns, and indicate connection. (Moreover, UML behavioral models cannot reference components.)

9.2 Connectors

There are two natural choices for representing connectors in this scheme: as dependencies between a component the ports/interfaces realized by a component (visually simple but lacking expressiveness), or as components themselves. If we represent connector instances as dependencies between components, we have the option of representing connector types as stereotypes, with consequences we addressed in previous sections. Unfortunately, although dependencies are visually appealing, the built-in dependency notion in UML does not adequately capture the idea of architectural connection or provide an explicit descriptive capability. Representing a connector as a UML component addresses

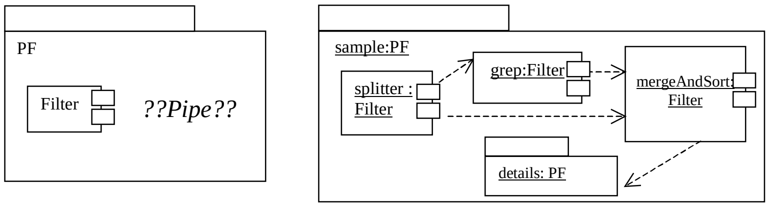

Fig. 5. Components as UML components.

this problem, but unfortunately blurs the distinction between components and connectors.

10 Using Subsystems

We now describe how a subsystem (a stereotyped UML package) can be used to describe the components in a system and the component types in a style. This approach has the appeal that packages are an ideal way to describe coarse-grained elements as a set of UML models. Also, the package construct is already familiar to UML modelers, and to those extending UML, as a way of bundling large pieces or views of a system.

In Figure 6, we describe the "filter" type as a package and filter instances as package instances (e.g., "splitter").

10.1 Components

The subsystem construct is used in UML to group or encapsulate a set of model elements that describe some logical piece of a system, similar to components in architectural descriptions. Subsystems (indeed, any package) can include structures based on any of the UML models. This has an advantage over describing components and connectors as classes – by identifying a component or connector with a package, we can not only include structure as classes (or objects), we can also include behavioral models. There are many options for describing component or connector substructure. This approach also has a visual appeal – substructure can be depicted as "embedded" in the package. Component and Component types would be modeled in essentially the same way, although one could also take advantage of the UML template mechanism when defining a type.

Although visually appealing, this approach suffers from a number of problems. In UML, a package is (semantically speaking) little more than a folder in which related models and elements can be stored. One does not talk about a package having interfaces; instead a package makes its elements (e.g., classes) available to other packages through export. Representing substructure (like ports) as elements contained by a package is also counterintuitive. The fact that certain elements correspond to ports, others to properties, others to reps, is misleading. Expressing the behavior of a component would also be awkward, since packages themselves do not have dynamic behavior in UML. Packages may only be related by dependence, which restricts how connection/attachment can be depicted. It also blurs the distinction between a system and a component in a system.

10.2 Connectors

There are two natural choices for representing connectors in this scheme: dependencies (visually simple but lacking expressiveness), or packages themselves. While dependencies have visual appeal, the dependency notion does not fully capture the notion architectural connection. As we've noted before, a package does not have to describe of connector properties and structure. As with components, we could include behavioral description of the connector within the package, although relating it meaningfully to the

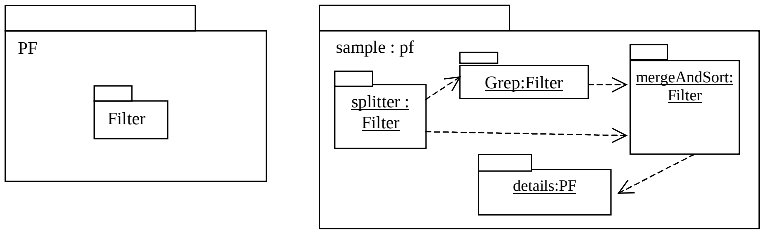

Fig. 6. Components as subsystems.

package itself is difficult, as it was for components.

11 Discussion and Conclusions

We have briefly examined five strategies for encoding in UML the representations of architectural structure expressible using modern ADLs: for each of these we considered a number of variations and considered the strategies with respect to completeness, legibility, and semantic match. What are the lessons to be gained from this?

First, *there is no single best way to encode ADLs in UML*. Each of the strategies has strengths and weaknesses, depending on how well they support the evaluation criteria. With respect to completeness and legibility there is a typically a tradeoff: encodings that emphasize completeness (by providing a semantic home for all of the aspects of architectural design) tend to be verbose, while graphically appealing encodings tend to be incomplete. Hence, the best strategy will depend on what aspects of architectural design needed to be represented. In restricted situations (for example, if there is only one type of connector) it may be preferable to use an incomplete, but visually appealing, encoding.

Second, *all of the encodings exhibit some form of semantic mismatch*. UML's vocabulary of classes, objects, packages, components, associations, etc., while varied and rich, is ultimately designed to support an object-oriented view of software design. As such, UML does not provide a completely adequate foundation for architecture-based description of systems. In this paper, we illustrated a number of specific examples of mismatch, including the following: (a) neither the class, subsystem or component concept is a perfect match to the ADL component concept; (b) unlike objects, architectural instances may need to define additional structure not defined by their types; (c) the port concept has no good analog in UML, since unlike interfaces, a port should be able to define both provided and *required* services, and a component might have multiple instantiations of a particular port type; (d) there is no satisfactory way to fully describe a connector and its roles; and (e) although the ADL type and instance concepts are very similar to the class and object concepts of UML, neither class diagrams nor collaboration diagrams are wholly appropriate for describing architectural configurations.

Given these observations, one might well ask whether there are reasonable alternatives to direct encoding of architecture in ADL. We see two plausible alternative paths:

1. Continue to use ADLs, but map to OO implementations. In this approach architecture description retains its own notations, but tools are provided to convert those descriptions to *lower-level* object notations in situations where the implementation is done in an object-oriented fashion.
2. Extend UML to include additional concepts for architectural modeling. This could be done by extending the UML meta-model, or by defining a profile for architectural design. Indeed, we can already see the inclusion of architectural notions, such as ports, in proposals for real-time extensions to UML [22,28]. Other proposed extensions would permit the definition of a class in terms of a nested collaboration diagram – providing both scoping of associations and automatic creation of substructure.

Both of these are promising avenues of future work. Additionally, we see considerable value in extending our examination of mappings to non-structural aspects of software architecture, such as behavior, performance, and reliability. As with structure, several authors have examined possible encodings, although not in a comparative study. Finally, to make more progress in reconciling architecture description with UML it will be important to consider the problem at a more formal level. In this paper we were forced to appeal to intuition regarding issues of semantic similarity. It would be desirable to have a more precise foundation on which to base this comparison.

12 Acknowledgements

We acknowledge the contributions of the members of the ABLE group at Carnegie Mellon, of the SEI Product Line Systems Program, and the members of the UML standards community who are working toward a proposal for architectural description in UML. We give special thanks to Bran Selic for guiding us in the UML standardization process and to Pedro Pinto for his work on encoding architectural descriptions in UML.

References

1. Allen, R.; Garlan, A. Formalizing Architectural Connection. *Proceedings of the 16th Intl. Conf. on SW Eng.*, 1994.
2. Binns, P.; and Vestal, S. Formal Real-Time Architecture Specification and Analysis. *Proceedings of 10th IEEE Wkshp on Real-Time OS and Sw*, May 1993.

3. Buschmann, F; Meunier, R.; Rohnert, H.; Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons, 1996.
4. Booch, G.; Rumbaugh, J and Jacobson, I. *The UML User Guide*. Addison-Wesley.
5. Booch, G.; Rumbaugh, J and Jacobson, I. *The UML Reference Manual*. Addison-Wesley.
6. Coglianese, L. and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. *In Proceedings of AGARD'93*, May 1993.
7. Garlan, D; Monroe, Robert T. and Wile, D. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Cambridge University Press, 2000.
8. Garlan, D.; Monroe, R.T. and Wile, D. *Acme Reference Guide*. Available from http://www.cs.cmu.edu/~acme.
9. Garlan, D.; Allen, R.; and Ockerbloom, J. (1994), Exploiting Style in Architectural Design Environments. *SIGSOFT'94*.
10. Garlan, D. and Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
11. Hofmeister, C., Nord, R.L. and Soni, D.. Describing Software Architecture with UML. *Proceedings of the TC2 1st Working IFIP Conf. on Sw Architecture (WICSA1)* 1999.
12. Kobryn, C. Modeling Enterprise Software Architectures Using UML. *In 1998 Proceedings International Enterprise Distributed Object Computing Workshop*, IEEE, 1998.
13. Kruchten, P.B. The 4+1 View Model of Architecture. *IEEE Software*, pp. 42-50, Nov. 1995.
14. Luckham, D.; Augustin, L.M.; Kenney, J.J.; Vera, J.; Bryan, D.; Mann, W. (1995) Specification and Analysis of System architecture using Rapide. *IEEE Trans on Software Eng.*
15. Magee, J.; Dulay, N.; Eisenbach, S. and Kramer, J. *Specifying Distributed Software Architectures. Proceedings of the 5th European Software Eng. Conf.*, 1995.
16. Medvidovic, N. and Rosenblum, S. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. *Proc. of the TC2 1st Working IFIP Conf. on Sw. Arch. (WICSA1)*, 1999.
17. Medvidovic, N. and Taylor, R.N (1997), A Framework for Classifying and Comparing Architecture Description Languages. *In Proceedings of the 6th European Software Engineering Conference together with with FSE4* .
18. Medvidovic, N.; Oreizy, P.; Robbins, J.E. and Taylor, R.N. (1996), Using object-oriented typing to support architectural design in the C2 style. *Proceedings of ACM SIGSOFT'96: 4th Symp. on the Found. of Software Eng. (FSE4).*
19. Monroe, R.T.; Kompanek, A; Melton, R. and Garlan, D. Architectural Style, Design Patterns, and Objects. *IEEE Software*, January 1997.
20. Moriconi, M.; Qian, X.; and Riemenschneider, R. (1995), Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pp. 356-372 (April).
21. Object Management Group (OMG), Analysis and Design Platform Task Force. White Paper on the Profile mechanism Version 1.0. OMG Document ad/99-04-97, April 1999.
22. OMG. UML Profile for Performance, Scheduling and Time. OMG Document ad/99-93-13.
23. OMG. UML Profile for Corba. RFP. OMG Document ad/99-03-11.
24. Robbins, R.E.; Medvidovic, D.F.; Redmiles, D.F. and Rosenblum, D.S. Integrating Architecture Description Languages with a Standard Design Method. *In Proceedings of the 20th Intl. Conf. on Software Eng. (ICSE'98).*
25. Rational Software Corporation and IBM (1997), OCL specification. OMG document ad/97-8-08. Available from http://www.omg.org/docs/ad.
26. UML Semantics. OMG ad/97-08-04. http://www.omg.org/docs/ad/97-08-04.pdf .
27. UML Notation Guide. OMG ad/97-08-05. http://www.omg.org/docs/ad/97-08-05.pdf
28. Selic, B; Rumbaugh, J. Using UML for Modeling Complex Real-Time Systems. White Paper.
29. Shaw, M.; DeLine, R.; Klein, D.; Ross, T; Young, D.; and Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Eng*, 1995.
30. Spitznagel, B and Garlan, D. Architecture-Based Performance Analysis. *Proceedings of the 10th Intl. Conf. on Software Eng. and Knowledge Eng. (SEKE'98*), 1998