To Appear: *Science of Computer Programming* Journal,
Special UML Edition, Elsevier Science, ~2001.

Reconciling the Needs of Architectural Description with Object-Modeling Notations

David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek

Carnegie Mellon University
Pittsburgh, PA 15213 USA

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, an 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 UML. Specifically, we describe (a) the principal 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 loosely 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 [24]. 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 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 for architectural descriptions? A number of authors have argued "yes," claiming that the current capabilities are precisely what is needed [7,19]. Moreover, recently there have been proposals that attempt to show how ADL concepts can be mapped directly into a notation like UML [17,22,23,41].

Unfortunately, each proposal is different, and each covers only *some* aspects of architectural description. We believe that what is needed is a 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.[1] 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,[2] 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, for example, proposes a set of (4+1) views—Logical, Process, Physical, Development, and Scenarios—to capture the various aspects of architectural information [19]. Young, et al. recognized the importance of an architecture standard as a business asset and helped IBM to develop an Architecture Description Standard (ADS) based on UML in 1998 [43]. ADS is expected to facilitate asset creation and reuse and hence reduce cost, risk, and time in development projects. Likewise, Hewlett Packard developed an architecture template based on UML to document software and firmware architecture. The templates prescribe class notations for "architectural" diagrams [30]. Other authors have examined the use of object-oriented notations for modeling architectural styles and patterns [7]. Finally, profiles have been proposed within the UML standards community for commercial modeling languages that include explicit architectural concepts, including a profile for CORBA [29] and real-time systems [34,36].

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. Hofmeister, Nord, and Soni discuss a scheme for describing four views of software architecture (conceptual, module, execution, and code) using UML stereotyped classes, packages, and components, related with associations and dependencies [17]. They

[1] Behavioral aspects of software architectures are also relevant, but beyond the scope of this paper.

[2] To be more precise we should say UML Version 1.4, the current version as of the writing of this paper.

note the strengths of UML in describing the static structure of architecture and its variability, and the deficiencies of UML in describing view correspondences, component ports, and dynamic aspects of structure. They express concerns about the wide range of semantics associated with the same notation in UML, and about the blurring of distinction between architecture and implementation by using UML. Medvidovic and Rosenblum describe their partial success at modeling C2-style architectures in UML [23]. Their effort highlights UML's lack of support for connectors and architectural styles. Monroe, et al., illustrate the difference of capabilities between software architecture design and object-oriented design, and conclude that neither approach subsumes the other [26] – both are appropriate at various times in the development process; both share some common notions and concepts; but both approaches provide different abstraction mechanisms and address different fundamental issues. Buschmann, et al. describe a number of architectural styles in the form of patterns, expressed in UML [7].

As we noted earlier, each of these adopts a particular form of modeling that exploits the constraints of the particular architectural domain, language or application. Unlike this paper, these efforts 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 Background

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. These diagrams are used to capture different views of a system's structure, including code-oriented views (e.g., representing usage dependencies between software modules), run-time views (e.g., representing the structure of a system as it would exist during execution), and deployment views (e.g., indicating how the system is mapped to a physical system of processors, memory units, and communication links) [9,16,18,19].

In an effort to put the architectural description of run-time views on a more solid notational and semantic footing, over the past decade a number of architecture description languages have been proposed [2,4,10,11,20,21,22,27,37]. While these languages (and their associated tool sets) differ in many details, a general consensus about the main ingredients of architectural description has emerged. Focusing on architectural structure, we take that core set of concepts as the starting point for this paper.[3] In this shared ontology there are five basic concepts: components, connectors, systems, properties, and styles.

- *Components*[4] 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

[3] These concepts are largely those found in Acme [13,14], xArch [42], and ADML [3].

[4] In this paper, when we refer to "components" we will mean components in the Architectural Description Language sense, as described in this paragraph, rather than the UML notion of component. When we need to refer to the latter—as in Section 4—we will qualify the term "UML component."

ports, each defining a point of interaction between a component and its environment. A component may have several ports of the same type. For example, a server may have multiple HTTP ports.

- *Connectors* represent interactions among components and correspond to the lines in box-and-line descriptions. They provide the "glue" for architectural designs, and therefore deserve explicit modeling treatment. 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 their own *internal* architectures. We will refer to these subsystems 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 [38]. 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 have an associated interaction protocol property.
- An architectural *Style* represents a family of related systems, and typically defines a design vocabulary of component, connector, port, role, binding, and property types, together with rules for composing instances of the types [15]. Examples include data-flow architectures based on graphs of pipes and filters, blackboard architectures, and N-tiered systems.

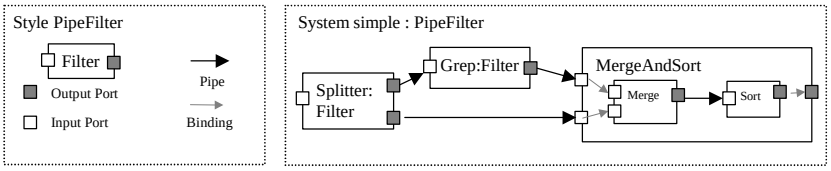

Fig. 1. A system in the *PipeFilter* style.

```
Family PipeFilter = {
    Port Type OutputPort;
    Port Type InputPort;
    Role Type Source;
    Role Type Sink;

    Component Type Filter;
    Connector Type Pipe = {
        Role src : Source;
        Role snk : Sink;
        Properties {
            latency : int;
            pipeProtocol: String = …;
        }
    };
};

System simple : PipeFilter = {
    Component Splitter : Filter = {
        Port pIn : InputPort = new InputPort;
        Port pOut1 : OutputPort = new OutputPort;
        Port pOut2 : OutputPort = new OutputPort;
        Properties { … }
    };
    Component Grep : Filter = {
        Port pIn : InputPort = new InputPort;
        Port pOut : OutputPort = new OutputPort;
    };
    Component MergeAndSort : Filter = {
        Port pIn1 : InputPort = new InputPort;
        Port pIn2 : InputPort = new InputPort;
        Port pOut : OutputPort = new OutputPort;
        Representation {
            System MergeAndSortRep : PipeFilter = {
                Component Merge : Filter = { … };
                Component Sort : Filter = { … };
                Connector MergeStream : Pipe = new Pipe;
                Attachments { … };
            }; /* end sub-system */
            Bindings {
                pIn1 to Merge.pIn1;
                pIn2 to Merge.pIn2;
                pOut to Sort.pOut;
            };
        };
    };
    Connector SplitStream1 : Pipe = new Pipe;
    Connector SplitStream2 : Pipe = new Pipe;
    Connector GrepStream : Pipe = new Pipe;

    Attachments {
        Splitter.pOut1 to SplitStream1.src;
        Grep.pIn to SplitStream1.snk;
        Grep.pOut to GrepStream.src;
        MergeAndSort.pIn1 to GrepStream.snk;
        Splitter.pOut2 to SplitStream2.src;
        MergeAndSort.pIn2 to SplitStream2.snk;
    };
}; /* end system */
```

Fig. 2. Textual description of the simple PipeFilter System.

To illustrate the use of these concepts, consider the simple example shown in Fig. 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, a *Pipe* connector type, and input and output Port types (*InputPort*/*OutputPort*). In addition, there would likely be constraints (not shown) that ensure, for example, that the reader/writer roles (*Source*/*Sink*) of a pipe (not explicitly shown in the figure) are associated with appropriate (*Input/Output*) ports of a filter. The system is described hierarchically: *MergeAndSort* is defined by a representation that is itself a *PipeFilter* system. Properties of the components and connectors might list, for example, performance characteristics used by a tool to calculate overall system throughput.

Fig. 2 shows a partial textual description of the *simple PipeFilter* system in Fig. 1, written in Acme, a typical ADL [14]. The description is in two parts. The first part, *Family PipeFilter*, defines a set of types (or style). The second part, *System simple*, uses these types to define a particular system. Each of the top-level component and connector instances has a corresponding definition containing its type, instance name, and interfaces. The attachments of ports to roles are also described explicitly. Omitted from the figure are properties of the elements (e.g., expressing the protocols of interaction for pipes), and details of the *MergeAndSort* representation.

This example highlights a number of key points about architectural description, as embodied by ADLs.[5] First, note that connectors are first class entities: they have types (e.g., *Pipe*) and they may have non-trivial semantics, for example, as defined by a protocol of interaction (elided above). Moreover, connectors have "interfaces," which identify their roles in the interaction, and may associate semantics with those interfaces. Thus, connectors can represent much richer abstractions than primitive interactions supported by a programming language (e.g., procedure call) [1].

Second, note that there may be many instances of the same type of architectural component or connector in a system description. Here we have several instances of the *Filter* and *Pipe* types. Also, note that different instances of a component or connector type may have quite different behavior: here we have five components of type *Filter*, although each performs very different kinds of computation. Using an ADL, differences in the component behavior would be indicated by different sets of properties of the instances.

Third, note that a given component may have several ports of the same type. For example, the *Splitter* filter has two output ports. Given that these architectural descriptions represent run-time system structure, it is important to permit multiple instances of the same port on a given component because they may have very different dynamic properties.

Fourth, note that bindings are not connectors. In this example the bindings serve to associate the input and output ports of the *MergeAndSort* filter with the input ports of *Merge* and the output port of *Sort* (respectively). The purpose of a binding is to provide a logical association – not a communication path – since a binding does not have any specific run-time behavior of its own.

4 UML Background

UML unifies a number of object modeling notations in a common framework and has become a standard object notation for industry [5,6,39,40]. While a detailed description of

[5] We will use the term "architectural description" to mean "architectural description as defined by ADLs."

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. We focus specifically on the structural elements classes, interfaces, collaborations, and components; the grouping elements packages and subsystems; relationships; and the extension mechanisms. Fig. 3 illustrates the graphical representation of most of the elements.

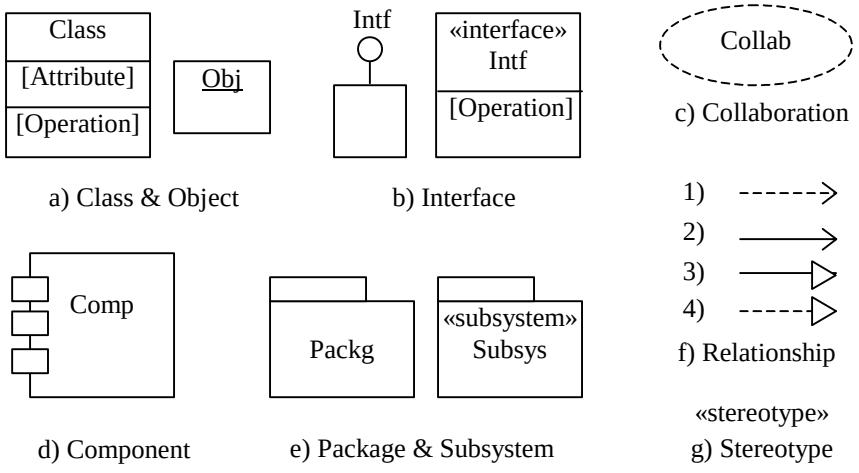

Fig. 3. UML model elements

- *Classes and Objects* (Fig. 3a): 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 are used in *class diagram*s to describe a static design view of a system. Classes have instances called *objects*, which may be included in *object/instance diagrams* to depict real or prototypical cases of the static design view, or used in models called *collaborations* to depict behavior under particular scenarios.[6] Classes can expose their functionality through a set of supported *interfaces*.
- *Interfaces* (Fig. 3b): Interfaces are collections of operations that specify the services of a class, component, or subsystem (see below).Each interface defines some aspect of an element's externally visible behavior as a set of operation specifications (i.e., signatures).
- *Collaborations* (Fig. 3c): Collaborations specify how classes, interfaces, and other elements cooperate to provide some aggregate behavior. Collaborations have structural, as well as behavioral, aspects. The structural aspect specifies how an element or operation is realized by a set of classifiers[7] and associations playing specific *roles*, while the behavioral aspect specifies the dynamics of the classifiers' interaction.
- *Components* (Fig. 3d): Components are used to describe the physical, deployable pieces of a system. Like classes, components in UML can expose their functionality through interfaces. Components are typically related to each other using dependency

[6] In UML, the prototypical "objects" in collaborations are referred to as collaboration *roles*.

[7] *Classifier* is the general term for elements that have structural and behavioral features. *Classes*, *components*, and *subsystems* are examples of classifiers.

relationships. The deployment of a system on a set of hardware elements is described by associating components with hardware *nodes*.

- *Packages and subsystems* (Fig. 3e): UML provides a grouping mechanism that is used to partition large UML models into manageable chunks called *packages*. Packages may contain structural elements, behavioral elements, and other grouping elements. UML also defines a special form of package called a *subsystem*, which is typically used to encapsulate the object models that define a coarse-grained module in a system. Subsystems are both packages and classifiers, and hence may have interfaces.
- *Relationships* (Fig. 3f): Model elements are related to one another with *dependencies*, *associations*, *generalizations*, and *realizations*. A dependency (Fig. 3f-1) is the most generic relationship in UML, indicating that an element depends in some way on the definition of another element. An association (Fig. 3f-2) is a richer relationship that describes an abstract relationship between classes (or, more generally, classifiers) and the roles the classes play in the relationship. Associations have instances called *links*, which connect objects in object/instance diagrams. An aggregation is a special kind of association that represents a structural relationship between a whole and its parts. A generalization (Fig. 3f-3) specifies a relationship in which objects of the specialized element can substitute for objects of the generalized element. A realization (Fig. 3f-4) is a semantic relationship in which one modeling element specifies a contract that another element guarantees to carry out.
- *Stereotypes* (Fig. 3g): To allow the extension of UML with domain-specific concepts, UML provides a mechanism for describing specialized forms of other model types. Typically a stereotype is defined by constraining the use or semantics of another model type.

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.

In addition to the kinds of model elements listed above, we should also mention the possibility of using *UML profiles*. A profile is a collection of stereotypes, constraints, tagged values, and notational conventions that can be bundled up to form a domain-specific language specialization of UML. Profiles are intended to serve particular subgroups of the object modeling community by providing a notation targeted to a particular domain or aspect of software development. For example, [6] describes a profile for Process Extensions. One could imagine creating an "architectural profile," or perhaps using an existing profile, such as the one for UML Real-Time [34] or SDL [33], as the carrier of architectural models.

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*. While such concerns are also relevant to architectural description, they are beyond the scope of this paper. However, to the extent that UML provides a natural home for associating behavior with architectures, in what follows, we'll note the possibility.

5 Strategies & Evaluation Criteria

We now consider four 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 (ports, connectors, systems, and representations).

Summarizing briefly, the four strategies (corresponding to the next four sections) are:

1. *Classes & objects*: Component types are represented by UML classes and component instances by objects;
2. *Classes & classes*: Component types are represented by UML classes (possibly using stereotypes), and component instances are also represented by classes;
3. *UML components*: Component types are represented as UML components and component instances as UML component instances;
4. *Subsystems*: Component types are represented as UML subsystems and component instances as subsystem instances;

We will also consider the UML Real-Time Profile [34] as a particular variant on these strategies, based on the mapping of ADL concepts to UML Real-Time described in [8].

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

6 Classes & Objects

Perhaps the most natural candidate for representing component types in UML is the class concept. Classes describe the conceptual vocabulary of a system, just as component types form a major part of the conceptual vocabulary of an architectural description in a particular style. Additionally, the relationship between classes and objects is similar to the relationship between component types and their instances: that is, the type/class defines what is common for each of its instances/objects. The mapping described for the C2 architectural style in [22] is a variation on this approach.

Fig. 4 illustrates the general idea for the example system introduced earlier. Here we characterize the *Filter* architectural type as the UML class *Filter*. Instances of filters, such as *Splitter*, are represented as corresponding objects in an object (instance) diagram. To provide a namespace boundary, we enclose the descriptions in packages. The representation of *MergeAndSort*, denoted *Details*, is shown as another package, and will be discussed later. Connectors are represented as associations in the class diagram, and links in the instance diagram. We now take a closer look at this strategy and some of its principal variants, considering how architectural components, ports, connectors, systems and representations can be represented in the following five subsections, respectively.

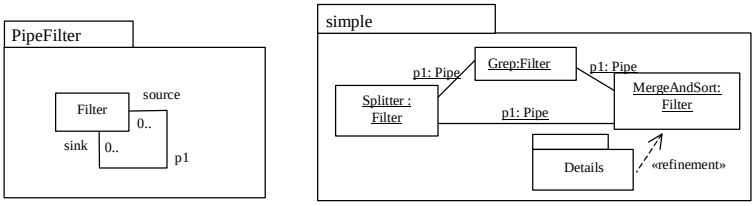

Fig. 4. Types as classes, instances as objects.

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. They support the full set of UML descriptive mechanisms for describing 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), or by attaching a different stereotype altogether.

The UML Real-Time Profile takes the latter approach to constrain, among other things, the visibility of operations and attributes of the instances of a class (Capsule-stereotyped) to explicitly provide for hierarchy and encapsulation [34].

While the use of classes and objects is a reasonable approach to representing architectural component types and instances in UML, there are some important semantic differences. In UML instances of classes are assumed to perform the same computations and to have the same internal structure. However, in ADLs instances of components may have very different behaviors and structure. In the example above, for instance, although *Splitter* and *MergeAndSort* are both of type *Filter* they obviously compute very different things, and may have different internal structure, as well as other quality attributes. Indeed, *MergeAndSort* has detailed architectural substructure, whereas *Splitter* does not.

To model the types of component instances such as the filters above, we either have to ignore the differences in those filters (by using a single type), or use five specialized filter classes to capture their differences – one for each component instance (cf., Section 7). The first choice is inadequate because it does not allow us to distinguish important differences between component instances. The latter is problematic because we end up with a proliferation of class definitions.

6.2 Ports (Component Interfaces)

There are several possible ways to represent ports. Fig. 5 illustrates some of the options.

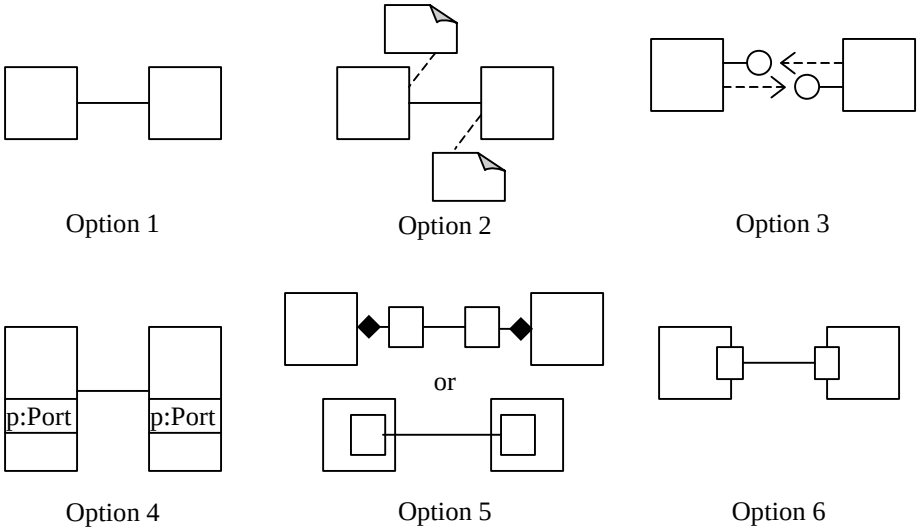

Fig. 5. Ways to represent ports

Option 1: No explicit representation.
We can leave ports out of the model entirely, the option we adopted in Fig. 4. 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, when the ports can be inferred from the system topology, or when the description is at a high enough level of abstraction that port information need not be represented.

Option 2: Ports as Annotations.
We can represent ports as annotations. This approach has the attraction that it is light-weight and can be used to specify arbitrary information about ports. But it has the disadvantage that annotations have no semantic value in UML, and hence cannot be used as a basis for analysis (such as checking the validity of connector attachments). As with Option 1, if the detailed properties of a port, or analysis of an architecture are not of concern, this might be a reasonable approach.

Option 3: Ports as Interfaces.
One seemingly obvious choice is to use UML interfaces to describe ports. 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 visual description of a port. In a collaboration diagram representing some instances of architectural elements, a UML association role qualified by the interface name provides a natural way to designate that a component instance is interacting through a particular port. 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. The most important difference is that in UML a given class (or object) can have only one

instance of a particular type of interface, whereas in ADLs a component may have many instances of the same port type. 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. This problem derives from the more fundamental problem that interfaces are primarily code-oriented entities, whereas ports are run-time entities. When one is only concerned about providing a set of methods that may be called by some other entity, it doesn't make sense to duplicate this information. But when one is concerned about the run-time points of interaction, one may want to have separate representations for each point of interaction, even though some may be providing the same set of services.

A second problem is that a UML interface is defined as a set of operations that can be invoked by other entities in the environment. 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. A third problem is that, unlike classes, interfaces do not have attributes or substructure. In contrast, many ADLs allow ports to be associated with arbitrary properties, and can even be "decomposed" into more detailed representations. For example some ADLs (such as Wright [2]) allow one to associate a protocol with a port.

Option 4: Ports as Class Attributes.
Ports can be treated as attributes of a class. In this approach ports are part of the formal structural model, essentially represented as named and typed slots of a class. While the types of attributes can in principal be any class, attributes are typically used only for primitive data values (numbers, strings, etc.). This restriction limits the expressiveness of this option.

Option 5: Ports as Contained Classes.
Another alternative is to describe ports as instances of classes that are *contained by* a component type. This approach overcomes the lack of expressiveness in the previous options, since we can now represent arbitrary port properties and substructure, and we can allow a component to have more than one port of the same type. Moreover, since they are represented as *contained* classes, port instances are created and destroyed with their containing object at run-time.

Unfortunately, as illustrated in Fig. 5, option 5 (top), this approach also suffers from problems: by representing ports as classes, we not only clutter the diagram, but we also lose visual 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 represented visually as contained classes (bottom part of Fig. 5, option 5). Unfortunately, it then becomes difficult to tell the difference between contained classes that are ports and contained classes that serve other purposes.[8]

Option 6: Ports as Classes with Special Conventions.
To address this last problem, one can adopt a specialized form of notation that places the contained classes representing ports on the boundary of the containing class. This is essentially the approach taken in the UML Real-Time Profile [34,36]. In that profile, classes designating components are termed "capsules." Interfaces to capsules, also called "ports," are represented as instances of Port-stereotyped classes and depicted as open or closed boxes at the class boundary.

A UML Real-Time port is required to have a type that corresponds to a specific role in some connector's interaction protocol, modeled separately using a collaboration associ-

[8] Another option is to represent ports as contained classes with interfaces. This is discussed in [12].

ated with that connector (see Option 2 in Section 6.3). These collaborations specify incoming as well as outgoing signals for each protocol role. This distinction corresponds well with the ADL notion of ports that specify both the services *provided* by the component as well as those it *requires*.

The disadvantage of this approach is that although the visual representation is intuitive, it is, graphically speaking, non-standard UML, and therefore requires specialized tools to achieve the desired visual capabilities and to exploit the distinction between normal contained classes and ports.[9]

6.3 Connectors

In an ADL, connectors are typically first-class, typed entities with potentially rich specifications. We now consider alternatives for representing connectors for the class-object strategy.

Option 1: Connector types as Associations; connector instances 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, which become instantiated as 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 (since both are depicted visually as connecting lines), 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 association 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: Connector types as Association Classes.
One partial solution to the lack of expressiveness is to qualify an association with a class that represents the connector type. In this way the attributes of a connector type or instance can be captured as attributes of the corresponding association 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 connector types are modeled as association classes, and where ports attached to the

[9] One such tool is Rational's Rose RealTime tool.

association ends are identified with classifier roles in a collaboration that defines the behavior of the connector [34,36]. The connector behavior is explicitly provided by a *protocol*, a stereotyped collaboration that specifies how the participating roles (played by the ports) interact. The advantage of representing a connector by an association class and a protocol is that the structural and behavioral specifications are made explicit. Unlike UML association classes, UML Real-Time connectors partially address the representation of connector roles. The stereotype of the UML Real-Time connector requires the association ends of the connector to correspond to the protocol roles of the connector's protocol, effectively making the association ends the connector roles. Unfortunately, association ends are not first-class entities in UML and, hence, the connector roles cannot have attributes and cannot be refined [8].

Option 3: Connector types as Classes; connector instances as Objects.
Another alternative is to give connectors first-class status by modeling connector types as classes, and connector instances as objects. In combination with Option 3 in Section 6.2, this is essentially the approach taken in [23] to make connectors first-class entities. We then have the same options for representing roles (connector interfaces) as we had for ports, as outlined in Section 6.2: (1) not at all, (2) as annotations, (3) as interfaces realized by a class, (4) as attributes, or (5) as classes contained by a connector class. 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. The main disadvantage of this approach is the loss of visual distinction between component types/instances as classes and connector types/instances as classes, since both are represented as classes/objects.

6.4 Systems

In an ADL a system is defined as a graph of component and connector instances. As such it defines an aggregating unit, and provides a boundary of abstraction around an interacting set of components and connectors. We now consider the options for representing systems in this class-object strategy.

Option 1: Systems as UML Subsystems.
The primary mechanism in UML for grouping related elements is the *package*. UML also defines a standard package stereotype, called *subsystem*, to group UML models that represent a logical part of a system. A subsystem is also a classifier, which means it acts like a class: it may be instantiated, support interfaces, etc. Taken primarily as a means of aggregating other parts, subsystems (or even packages) make a reasonable choice for modeling ADL systems.

Option 2: Systems as Contained Objects.
A second option is to use object containment to represent systems. Components are represented as instances of contained classes, and connectors are modeled using one of the options outlined above. Objects provide a strong encapsulation boundary, and carry with them the notion that each instance of the class will have the associated "substructure."

However, there are problems with this approach. Most importantly, the associations (e.g., used to model connectors) between contained classes are not scoped by the class. That is, it is not possible for one to say that a pair of classes interacts via a particular connector (modeled as an association) *only* in the context of a particular system. So for

example, if I indicate that two contained classes interact via some association, that association is also valid for instances of those classes used anywhere else in the model.[10]

Option 3: 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. A collaboration defines a set of participants and relationships that are meaningful for a given set of purposes, which, in this case, is to describe the run-time structure of the system. The participants define *classifier roles* that *objects* play (conform to) when interacting with each other. Similarly, the relationships define *association roles* that *links* must conform to.

Collaboration diagrams can be used to present collaborations at either the specification or the instance level. A specification-level collaboration diagram shows the roles (defined within the collaboration) arranged in a pattern to describe the system substructure. An instance-level collaboration diagram shows the actual objects and links conforming to the roles at the specification level, and interacting to achieve the purpose. Therefore, a collaboration presented at the instance level is best used to represent the run-time structure of the system.

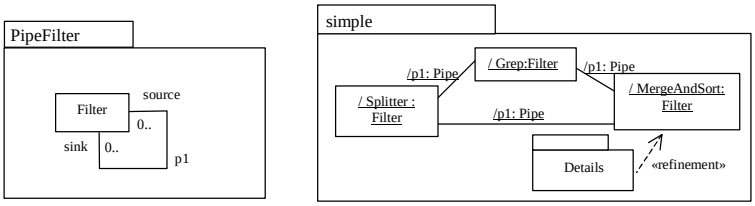

Fig. 6. Systems as Collaborations.

Fig. 6 illustrates this approach. The *Filter* architectural type is represented as in the previous section. Instances of *Filters* and *Pipes* are represented as corresponding classifier roles (e.g. "*/Splitter*" indicates the *Splitter* role) and association roles, and the objects and links conforming to those roles are shown in the collaboration diagram at the instance level (indicated by underlines on the names).

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.

Option 4: Systems as Collaborations with Special Conventions.
While the previous option explored collaborations at the instance level, using collaborations at the specification level is another alternative for describing systems. This is the approach taken by the UML Real-Time Profile, in which the structural decomposition of a capsule, equivalent to the architectural representation of a component, is described using a collaboration diagram.

[10] Recent proposals for UML 2.0 revisions suggest that this limitation may be eliminated in the future by introducing a new notion of structured classes. In that scheme an class could be defined in terms of a set of contained classes whose associations are scoped by the containing class.

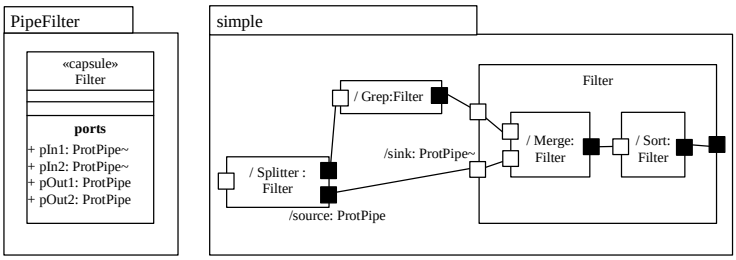

Fig. 7. UML Real-Time collaboration diagram for system *simple*: *PipeFilter*.

Fig. 7 shows the simple pipe-filter system of Fig. 1, but now drawn in the UML Real-Time notation. In Fig. 7, the filters become capsules of type *Filter*, each with input and output ports. The pipes in the ADL diagram become connectors that conform, in this case, to a pipe protocol (*ProtPipe*) with a *source* and a *sink* protocol roles. The output and input ADL ports, joined by the connector, therefore play the *source* and *sink* protocol roles, respectively.

The visual similarity of Fig. 7 to the ADL architecture diagram is apparent. This approach is visually intuitive and takes advantages of the UML Real-Time capsule provision for encapsulation and decomposition, the UML Real-Time port support for compact notation and for defining multiple points of interactions of the same type, and the UML Real-Time connector support for explicit behavioral specification.

6.5 Representations

In an ADL a component (or connector) representation is just a system that provides a more detailed structural view of a component or connector. To model a representation two things are needed. First, one must be able to associate the more detailed description with the element it is describing. Second, one must describe a set of bindings that indicate how the external interfaces of the component or connector are realized by the internal interfaces of the representation.

When systems are represented using UML subsystems (Option 1 in Section 6.4), an architectural element can be related to its representations through a dependency relationship (as illustrated in Fig. 4), possibly with an annotation indicating the name of the representation. Another option is to associate a representation with the element it is representing via a realization relationship, which is used to relate a specification to an implementation. This relationship captures the idea that a representation of a component (or connector) should realize the external behavior of the represented component (or connector). However, in either case it becomes problematic to describe the specific bindings between internal and external interfaces. When systems are represented using contained objects (Option 2 in Section 6.4), the association between the representation and the element it is representing is already apparent via the containment relationship.

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); and (3) as set of elements with attributes that describe the bindings. Alternatively, if systems are represented as classes, representations can be associated with their parent element using associations and links as bindings. UML –Real-Time adopts the latter approach, essentially modeling a binding as a kind of connector. This approach has the disadvantage that

it is not possible to distinguish binding relationships (which associate functionality, but imply no run-time mechanism) from connectors (which typically indicate a run-time communication mechanism).

6.6 Summary

To summarize, the strategy of representing architectural component types as classes, and component instances as objects has a number of advantages. It naturally captures the type-instance relationship, and it gives an expressive home for describing the properties of component types. However, it suffers from the problem that component instances of the same type must have identical behavior (not required in some ADLs). With respect to the other architectural elements there were a number of options. The most expressive way of representing ports is to treat them as contained classes. However, to avoid visual clutter it becomes important to introduce visual conventions, such as those in the UML Real-Time Profile. Connectors also have several representational options, although the most natural is to use associations. This option preserves the semantic distinctions between components and connectors, but the use of associations makes it difficult to define connector types as semantically-rich, stand-alone entities. This problem can be partially overcome by associating a collaboration with an association class to define its behavior, as is done in UML Real-Time.

7 Classes & Classes

The second class-based strategy is to represent component types as classes (like the first strategy described in Section 6) and also component instances as classes. By representing both component types and instances as classes, we have the full set of UML features to describe both component types and instances. This is the approach used in [32].

Fig. 8 illustrates this strategy, defining both the *Filter* type and instances of *Filters* (e.g., *Splitter*) as classes. We now examine this strategy in light of the previous class-based strategy.

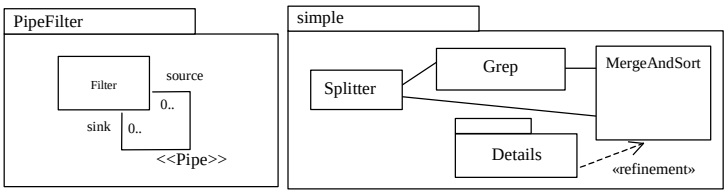

Fig. 8. Types & instances as classes.

7.1 Components

Representing instances as classes has a number of advantages. First, we can capture a set of possible run-time configurations (e.g., one-to-many relationships) in a single diagram. Second, and more importantly, by using classes, we can model the fact that different component instances of the same type have different behavior, substructure, and properties. This overcomes the limitation of the class/object approach mentioned in Section 6.1 where we either had to ignore differences in the different filters of our example, or give each a different component type.

Although the class-class strategy has many of the strengths of the first (class-object) strategy, it suffers from a number of problems. Representing both types and instances as classes obviously blurs the distinction between type and instance, an unfortunate consequence. However, the major problem with this approach is that, due to the semantics of classes, this approach is unable to handle situations in which there are multiple instances of the *same* component type. Consider the system description (a1) in Fig. 9. Although visually it suggests that the system contains two distinct instances of component A, there is, in fact, only *one* instance, since both (a1) and (a2) are equivalent diagrams in UML. On a related note, it is also worth observing that the class diagram (b1) in Fig. 9 does not require A to be shared in the object instance level. Either of the instance diagrams in (b2) are legal representations of (b1).

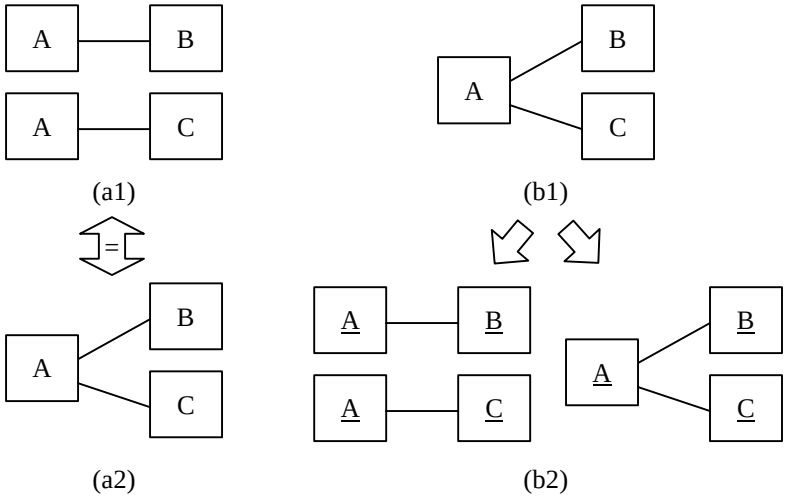

Fig. 9. Semantic problems with instances as classes

Using the class-class scheme a component type is related to its instances using generalization. That is, component instances are subclasses of their corresponding component types. For example, we could define a *Filter* class and have each of its instances (e.g., *Splitter, Grep*, etc.) be subclasses. However, as noted, this approach does not capture the type-instance distinction that is inherent in the relationship between component types and instances.

One alternative that allows us to distinguish classes that are meant to represent component types from those that represent component instances is to use UML stereotypes. 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.

To illustrate, in Fig. 10, 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.

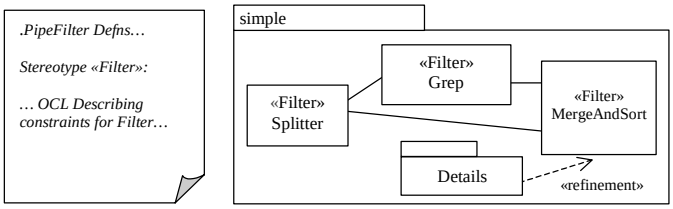

Fig. 10. Types as stereotypes, instances as stereotyped classes.

The observations that we made previously about the suitability of classes to represent component instances apply here as well. 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 subtyping 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 of the other built-in stereotypes to add meaning to a component that already has a stereotype.

7.2 Ports, Connectors, Systems, and Representations

The choices for representing ports, connectors, systems, and representations are similar to those of the first strategy.

8 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 Fig. 11 we represent the *Filter* type as a UML component and instances (e.g., *Splitter*) as instances of this UML component.

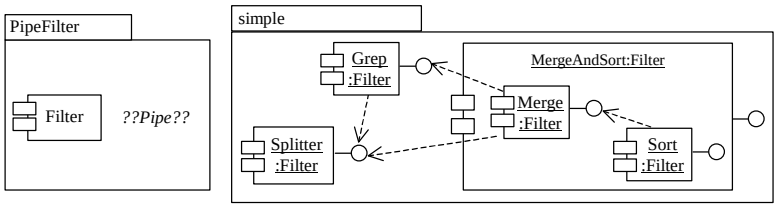

Fig. 11. Components as UML components.

8.1 Components

In some ways UML components are a natural candidate for representing architectural components. UML components have interfaces, may be deployed on hardware, and

commonly carry a stereotype depicted with a custom visualization. UML components are often used as parts of diagrams that depict an overall topology, and just as it is natural to talk about mapping architectural components to hardware, UML 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, 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 a deployable piece of a system. Nonetheless, the concepts share more than a name. UML components expose interfaces (as with classes) that can be used to represent the ports exposed by an architectural 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 UML components, limiting how we can describe ports, represent patterns, and indicate connection.

8.2 Connectors

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

9 Subsystems

We now describe how UML subsystems can be used to describe the architectural components in a system and their component types. This strategy has the appeal that subsystems 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 Fig. 12, we describe the *Filter* type as a subsystem and *Filter* instances as subsystem instances (e.g., *Splitter*).

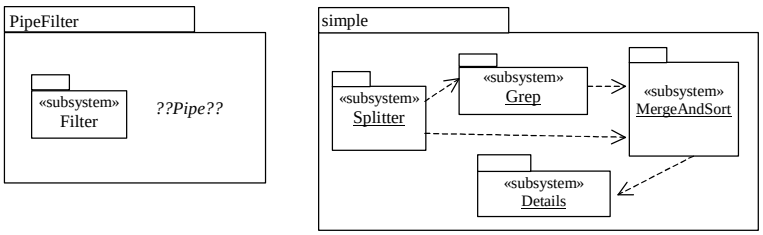

Fig. 12. Components as subsystems.

9.1 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. There are two types of subsystems—*open* and *closed*. An open subsystem allows direct external interactions with public elements contained within it. A closed subsystem is treated like a black box, and all interactions occur directly with the subsystem [31]. Clearly, a closed subsystem offers the necessary encapsulation provided by architectural components.

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 modeling a component or connector as a UML subsystem, not only can we include structure as classes (or objects), we can also include behavioral models. In addition, as classifiers, subsystems can also define interfaces (the lollipop notation). This approach also has a visual appeal – substructure can be depicted as being "embedded" in the subsystem.

However, the use of subsystems to model components suffers from a number of problems. In UML, since a subsystem has no behavior of its own, all communications sent to a closed subsystem must be redirected to instances inside the subsystem, and UML leaves that redirection unspecified as a semantic variation point. Secondly, subsystem interfaces raise the same set of issues mentioned for class interfaces. (That is, it is impossible to model several interfaces of the same type on the same subsystem.) Third, representing substructure (like ports) as elements contained by a subsystem is arguably counterintuitive. The fact that certain elements correspond to ports, others to properties, others to representations, is likely to be misleading.

9.2 Connectors, Systems, and Representations

There are two natural choices for representing connectors in this scheme: as dependencies or as subsystems themselves. Using dependencies is visually appealing, but dependencies do not provide a way to define more detailed aspects of a connector. Using subsystems to model connectors is similar to using objects or classes to model connectors in the previous two approaches, and suffers from the problem that components and connectors are not distinguishable.

Given the use of subsystems to represent components, it is also natural to use them to encapsulate whole systems, as well as to capture substructure.

10 Discussion and Conclusions

In this paper we examined four main strategies for encoding in UML the architectural elements typically found in modern architectural description languages. Centered on the choice for representation of architectural component types and component instances, those strategies were (1) classes and objects, (2) classes and classes (possibly using stereotypes), (3) UML components, and (4) subsystems. For each of these we considered a number of variations for handling the other architectural elements (connectors, ports, systems, and representations), and examined the strategies with respect to completeness, legibility, and semantic match. What are the lessons to be gained from this?

First, it is clear that the current definition of UML does not favor a single best way for encoding architectural concepts. Each of the strategies has certain strengths and weaknesses, depending on how well they support the evaluation criteria. With respect to

completeness and legibility there is 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. Moreover, the choice may depend on what aspects of an architecture one cares about modeling. For example, if it is important to model the behavior of connector mechanisms, certain choices will be favored over others.

Second, all of the encodings exhibit some form of semantic incompleteness or mismatch. As we have illustrated, the key stumbling blocks are difficulties in faithfully representing one or more of the following architectural concepts:

a. *Ports*. There were two issues. One is the need to have multiple ports of the same type. The other is to be able to associate a variety of information with a given port, such as its protocol of interaction, assumptions about its environment, performance properties, etc.
b. *Connectors*. The primary difficulty here is to support definition of connector types independent of any particular use of them. A secondary problem is the ability to associate semantic properties with a connector, such as its protocol of interaction, or properties. A third difficulty is the ability to represent the interfaces of connectors explicitly.
c. *Substructure*. Because complex architectural descriptions require hierarchical descriptions it must be possible to define an architectural component (or even connector) in terms of its more detailed substructure. The main problems with achieving this were (a) finding a way to limit the scope of the substructure definition to the element being elaborated, and (b) finding a way to explain how the internals map to the external interfaces.

With respect to solving these problems, the class-object and subsystem strategies appear to be the best solutions (although neither is perfect). Moreover, the UML Real-Time Profile includes certain specialized forms of the class-object strategy that further improve the usability of that strategy.

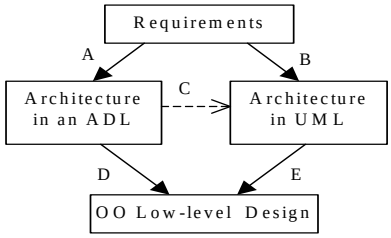

Fig. 13. Refinement paths.

Given these difficulties, one might well ask whether there are reasonable alternatives for making progress in the future. We see two plausible alternatives:

1. Use an ADL for architectural description, but provide tools that help refine these descriptions into lower-level, object-oriented designs in situations where the implementation is done in an object-oriented fashion. Referring to Fig. 13, we follow path A-D, rather than A-C-E (or B-E, if we are staying completely within an object-oriented universe).

2. Extend existing UML modeling elements to better support architectural modeling. Two obvious places where enhancements could help are (a) allowing classes to be described in terms of substructure that is scoped within the class boundaries, and (b) clarifying the way in which subsystems can function as both packages and classifiers [25]. Discussion is currently underway for doing exactly this in future versions of UML.

Both of these are promising avenues of future work. In fact, Option 6 of Section 6.2, Option 2 of Section 6.3, and Option 4 of Section 6.4 attempted to show partial work along these two paths, by using UML –Real-Time, which contains architectural notions, and by providing a mapping strategy to convert an ADL model to an object-oriented model. Additionally, we see considerable value in extending our examination of mappings to non-structural aspects of software architecture, such as behavior, and other properties like performance, and reliability. 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 nice to have a more precise foundation on which to base this comparison.

11 Acknowledgements

This material is based upon work sponsored by the Defense Advanced Research Projects Agency (DARPA) and supported by the Air Force Research Laboratory under Contract No. F30602-00-2-0616 as well as grants from Lucent Corporation and Daimler Chrysler Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the United States Air Force, Lucent Corporation, or Daimler Chrysler Corporation.

We acknowledge the contributions of the members of the ABLE group at Carnegie Mellon, of the SEI Product Line Systems Program, and members of the UML standards community—Bran Selic, Birger Møller-Pedersen, Thomas Weigert, Oystein Haugen, James Rumbaugh, Joaquin Miller, and others. We also would like to acknowledge Pedro Pinto for his initial work on encoding architectural descriptions in UML, as well as the anonymous reviewers of earlier versions of this paper. This work is based on previous publications in the UML 2000 Conference [12] and the PDPTA 2001 Conference [8].

12 References

1. Allen, R. J. and Garlan, D. A Formal Basis for Architectural Connection. *ACM Trans. on Software Eng. and Methodology*, vol. 6, no. 3, Jul. 1997.
2. Allen, R. J. and Garlan, D. Formalizing Architectural Connection. *Proceedings of the 16th Intl. Conf. on SW Eng.*, 1994.
3. Architecture Description Markup Language (ADML). The Open Group. http://www.opengroup.org/tech/architecture/adml/adml_home.htm
4. Binns, P. and Vestal, S. Formal Real-Time Architecture Specification and Analysis. *Proceedings of 10th IEEE Workshop on Real-Time OS and Software*, May 1993.
5. Booch, G., Rumbaugh, J and Jacobson, I. *The UML User Guide*. Addison-Wesley, 1999.
6. Booch, G., Rumbaugh, J and Jacobson, I. *The UML Reference Manual*. Addison-Wesley, 1999.
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons, 1996.
8. Cheng, S. W. and Garlan, D. Mapping Architectural Concepts to UML-RT. *Proceedings of the Parallel and Distributed Processing Techniques and Applications Conf.*, Jun. 2001.

9. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. *Software Architecture Documentation in Practice* (To Be Published). Addison Wesley, 2001.
10. Coglianese, L. and Szymanski, R DSSA-ADAGE: An Environment for Architecture-based Avionics Development. *Proceedings of AGARD'93*, May 1993.
11. Garlan, D., Allen, R. and Ockerbloom, J. (1994), Exploiting Style in Architectural Design Environments. *SIGSOFT'94*.
12. Garlan, D. and Kompanek, A. J. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Proceedings of the Third Int'l Conf. on the Unified Modeling Language*, Oct. 2000.
13. Garlan, D., Monroe, R. T. and Wile, D. Acme: An Architecture Description Interchange Language. *Proceedings of CASCON 97, Toronto, Ontario*, Nov. 1997.
14. Garlan, D., Monroe, R. T. and Wile, D. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Cambridge University Press, 2000.
15. Garlan, D. and Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
16. Hofmeister, C., Nord, R. L. and Soni, D. *Applied Software Architecture*. Addison Wesley Longman, 2000.
17. Hofmeister, C., Nord, R. L. and Soni, D. Describing Software Architecture with UML. *Proceedings of the TC2 1st Working IFIP Conf. on Software Architecture (WICSA1)*, 1999.
18. IEEE Draft for Standard, *IEEE P1471 Draft Recommended Practice for Architectural Description*, Oct. 1999.
19. Kruchten, P. B. The 4+1 View Model of Architecture. *IEEE Software*, pp. 42-50, Nov. 1995.
20. Luckham, D., Augustin, L. M., Kenney, J. J., Vera, J., Bryan, D. and Mann, W. Specification and Analysis of System Architecture using Rapide. *IEEE Trans. on Software Eng.*, 1995.
21. Magee, J., Dulay, N., Eisenbach, S. and Kramer, J. Specifying Distributed Software Architectures. *Proceedings of the 5th European Software Eng. Conf.*, 1995.
22. Medvidovic, N., Oreizy, P., Robbins, J. E. and Taylor, R. N. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. *Proceedings of ACM SIGSOFT'96: 4th Symp. on the Foundation of Software Eng. (FSE4)*, 1996.
23. 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 Software Architecture (WICSA1)*, 1999.
24. Medvidovic, N. and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Software Eng.*, vol. 26, no. 1, 2000.
25. Miller, J. and Wirfs-Brock, R. How can anything be both a classifier and a package? *Proceedings of the Second Int'l Conf. on the Unified Modeling Language*, Oct 1999.
26. Monroe, R. T., Kompanek, A, Melton, R. and Garlan, D. Architectural Style, Design Patterns, and Objects. *IEEE Software*, Jan. 1997.
27. Moriconi, M., Qian, X. and Riemenschneider, R. Correct Architecture Refinement. *IEEE Trans. on Software Eng.*, pp. 356-372, Apr., 1995.
28. OMG. UML Profile for Performance, Scheduling and Time. OMG Document ad/99-93-13.
29. OMG. UML Profile for CORBA. RFP. OMG Document ad/99-03-11.
30. Ogush, M. A., Coleman, D., and Beringer, D. A Template for Documenting Software and Firmeware Architectures. Hewlett Packard, 2000.
31. Övergaard, G and Palmkvist, K. Interacting Subsystems in UML. *Proceedings of the Third Int'l Conf. on the Unified Modeling Language*, Oct 2000.
32. Robbins, R. E., Medvidovic, D. F., Redmiles, D. F. and Rosenblum, D. S. Integrating Architecture Description Languages with a Standard Design Method. *Proceedings of the 20th Intl. Conf. on Software Eng. (ICSE'98)*, 1998.
33. SDL Combined with UML, Recommendation Z.109, International Telecommunications Union, 1999.
34. Selic, B. UML-RT: A profile for modeling complex real-time architectures. Draft, ObjecTime Limited, Dec. 1999.

35. Selic, B., Gullekson, G. and Ward, P. T. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
36. Selic, B. and Rumbaugh, J. Using UML for Modeling Complex Real-Time Systems. White Paper, Mar. 1998..
37. 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.
38. Spitznagel, B and Garlan, D. Architecture-Based Performance Analysis. *Proceedings of the 10th Intl. Conf. on Software Eng. and Knowledge Eng. (SEKE'98*), 1998
39. UML Notation Guide. OMG ad/97-08-05. http://www.omg.org/docs/ad/97-08-05.pdf .
40. UML Semantics. OMG ad/97-08-04. http://www.omg.org/docs/ad/97-08-04.pdf .
41. Weigert, T., Garlan, D., Knapman, J., Møller-Pedersen B. and Selic, B. Modeling of Architectures with UML. *Proceedings of the Third Int'l Conf. on the Unified Modeling Language*, Oct. 2000.
42. xArch. Institute for Software Research, UC, Irvine. http://www.ics.uci.edu/pub/arch/xarch/ .
43. Youngs, R., Redmond-Pyle, D., Spaas, P. and Kahan, E. "A standard for architecture description." *IBM Systems Journal*, vol. 38, no. 1, 1999.