Submitted for publication.

Semantic Issues in Architectural Refinement

Tony Tseng, Jonathan Aldrich, David Garlan, and Bradley Schmerl
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
{tttseng,aldrich,garlan,schmerl}@cs.cmu.edu

Abstract

Relating software architecture to implementation is essential for effective software development and evolution. However, significant gaps exit between abstract architecture models and the more concrete models supported by implementation tools, making it difficult to ensure that an architecture and implementation are consistent.

In this paper, we characterize three key semantic challenges in refining an abstract architectural view to a more concrete one: mapping typing relationships, refining connectors, and handling information overlap. We outline general strategies for addressing these challenges, and describe a concrete tool that enables architects to make the transition from abstract to concrete architecture more effectively.

1. Introduction

It is well known that multiple views are necessary for capturing all interesting aspects of a software architecture [6]. Following the process outlined in Figure 1, an architect may begin with a highly abstract, component-and-connector-based view, which is refined into a more concrete view with component interfaces specified and an implementation strategy defined. The system is then implemented in some language, leading to yet another view based on the module structure of the source code. Other views may also be useful for performing various kinds of architectural modeling and analysis.

Although the views shown in Figure 1 are clearly related, there is rarely a perfect correspondence between them. The resulting semantic gaps can cause serious problems in the implementation and evolution of software systems if the program as built does not conform to the architecture as designed.

Recently, a number of projects have attempted to help bridge these semantic gaps by capturing architectural components and connectors at the implementation level. One such project is ArchJava, a language that integrates a specification of a run-time component-and-connector architecture into the static source code of a program [2]. Another example is UML 2.0, which contains explicit modeling features for architectural components and connectors, and supports source code generation from these models [21].

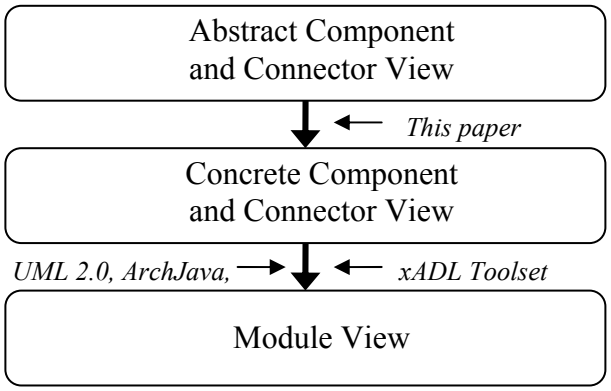

Figure 1. Three views of architecture, with the most abstract at the top and the most concrete at the bottom. This paper focuses on the transition from an abstract to a concrete component-and-connector view, while other work has focused on connecting a concrete view to an implementation-based module view.

Although these techniques can bridge the gap between a concrete component-and-connector (C&C) architectural view and source code, difficult issues remain in the gap between an abstract C&C view and the more concrete views that are used in the projects described above. This paper makes the following contributions to addressing these issues:

- In the next section, we use a running example to characterize three key semantic challenges in refining an abstract architectural view to a more concrete one: mapping typing relationships, refining connectors, and handling information overlap.
- In Section 3, we outline general strategies for addressing each of these challenges.
- In Section 4, we describe a tool that aids in refining abstract architectures to more concrete architectures.

Section 5 discusses related work, and Section 6 concludes the paper.

2. Refinement Challenges

Although they differ in the level of abstraction, abstract and concrete component and connector views contain many of the same elements, and we initially expected that mapping between these levels would be straightforward. However, when we began designing a tool to aid architects in mapping an abstract view to a more concrete one, we were surprised to find a number of hard technical challenges. For example, we found that it is often necessary to change typing relationships when refining an architectural design. Furthermore, more concrete architectural views may lack support for design-level information including connector representations, architectural styles, and properties, raising the question of how to represent this information at the concrete level.

In this section, we begin by defining more precisely what we mean by abstract and concrete architectural views. Next, we describe an abstract architecture that will be used as a running example for the rest of the paper. We then discuss each of the three key refinement challenges—and corresponding refinement strategies—that we identified: mapping typing relationships, refining connectors, and handling partially overlapping information between views.

2.1. Abstract and Concrete Views

Abstract Views. An *abstract component and connector view* is an architectural view of the components and connectors in a system that is focused on design-level constraints. In order to permit a high level of abstraction, and to maximize design-level reuse, this view avoids committing to an implementation strategy for the individual components, connectors, and interfaces between them. An abstract architectural view may have high-level specification information that has no direct analog at a more concrete level, such as non-functional requirements, the order of architectural events, or style information.

Abstract architectural views may also be *incomplete* specifications in that they focus on issues that are of particular interest at the design level. For example, an abstract view may represent only the main datapath through a system, abstracting away other possible communication paths.

Examples. Architecture description languages based on process algebras provide fairly abstract views of architecture, where interfaces are often represented as a collection of events with ordering constraints [4]. The SADL language supports refinement from abstract designs, such as a generic connector, to more concrete implementation choices, such as a shared variable implementing that connector [16].

Acme and xADL 2.0 are generic architecture description languages that have the capability of modeling architecture at both high and low levels of abstraction [7][10]. Both systems are extensible, allowing architects to model arbitrary architectural properties. Acme also provides language and tool support for architectural styles that can be freely composed together to form new styles.

Concrete Views. A *concrete component and connector view* is an architectural view of the components and connectors in a system that expresses the same conceptual architecture of the more abstract view, but commits to implementation strategies for each of the components and connectors. This view may omit non-functional requirements, not because they are unimportant but because they cannot be directly implemented. Unlike an abstract C&C view, a concrete view is typically complete in that it specifies all of the components and all of their interfaces.

Relation to Module Interconnection Views. Concrete component and connector views of an architecture may appear to be similar to a module interconnection view of source code, but they differ in that the components in a concrete C&C view are potentially run-time objects, not static source code modules, and connectors may be richer than simple module bindings. In practice, a run-time component may be implemented by many modules or by part of a module, and more than one component of the same component type (and with the same implementation code) may exist in the system. Furthermore, connectors may perform tasks such as buffering or network communication. Thus, the mapping between architectural elements in a concrete C&C view and modules in a module interconnection view may not be one-to-one.

The OMG's Model-Driven Architecture (MDA) also defines two different architectural views: platform-independent and platform-dependent [21]. In our taxonomy, both of these views generally share characteristics with concrete component and connector views or module interconnection views, depending on if they focus on component and connectors or code modules. This paper is thus concerned with refinement at a higher level of abstraction compared to the refinements discussed in the MDA.

Examples of Concrete Views. Concrete component and connector views can be modeled in a number of architecture description languages; ADLs that provide tool support for generating code from concrete C&C views include xADL 2.0, ArchJava,[1] and UML 2.0 [2][7][21]. The code generation tools require specifications of com-

[1]Since ArchJava is a complete language, it technically provides a compiler rather than a code generator.

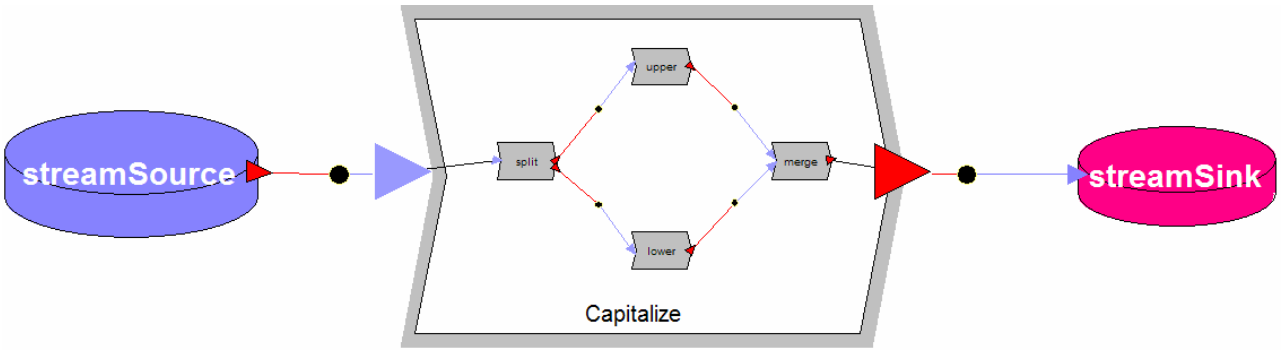

Figure 2. A Simple Pipe & Filter Abstract Architecture.

ponent interfaces that are complete and similar to those found in implementation languages.

Despite their concreteness, these languages all support component and connector views rather than (or in addition to) simple module interconnection views. Components in these views represent run-time objects with state and functionality, and which may be instantiated multiple times during program execution, in contrast to static code modules. These languages also support user-defined connectors with rich functionality.

2.2. Example Abstract Architecture

Throughout this paper we will refer to an example of an abstract component and connector view for a pipe and filter system, presented in Figure 2. This architecture consists of three components (a data source and sink, and a filter component called Capitalize), and two pipes connecting them. The Capitalize component is decomposed into a sub-architecture consisting of another pipe and filter system. The pipes have properties associated with them, such as buffersize and throughput. The roles of the pipes specify the protocols of interaction that the pipe understands. We desire to refine this architecture into an implementation. As mentioned above, this may seem straightforward, but there are a series of refinement decisions that need to be made to relate this abstract architecture to an implementation.

2.3. Challenge: Types

Types are an important mechanism for achieving design-level reuse at both the abstract and concrete levels of architecture. These two levels, however, may use types in quite different ways according to the intended modeling purpose of each level.

Concrete architecture representations, following conventional implementation languages, typically use types to characterize the interfaces of a component. These interfaces state what functions are used in inter-component communication, as well as what data is passed between these functions. In many cases, these interfaces are expected to be complete (including all relevant functions), both in order to guide implementors and to form the basis for a sound implementation-level type system.

In contrast, an abstract architectural view is more likely to use types to organize component specifications. These specifications are generally focused on issues of particular interest to the architect, and may therefore omit interface details that are irrelevant to these issues. Although component interfaces can be of interest at the abstract architectural level, many abstract representations focus more on the ordering of communication events or on non-functional properties.

Type Mappings. Because of these differences in emphasis, the type structure of an abstract architectural view may be quite different from that of a corresponding concrete view. Thus, component and connector types may not be in a one-to-one correspondence, complicating the challenge of refining an abstract view into a more concrete one.

An abstract architecture may not attempt to characterize the full behavior of its components, and therefore may use a single component type to describe two components that have significant differences at the implementation level. In Figure 2, for example, the upper, lower, split, and merge components are all modeled with the Filter component type.

A concrete architectural view may want to represent these as four distinct types, both in order to capture differences in interface or behavior and to express an implementation strategy that uses different modules for the two components' code. For example, an implementation in Java would likely use different classes to represent the upper and lower components, because these have different behavior. The converse is also possible, although probably less likely: a single concrete implementation may be flexible enough to implement two different abstract component types.

Type System Structure. In addition to differences in the relationships modeled between types, abstract and concrete ADLs may have type systems with different structure.

Concrete ADLs, following implementation languages, often use conventional type systems that focus on implementation-level substitutability. A crucial characteristic of conventional type systems is that all the external services a component requires are stated in its type, along with a subset of the services that the component provides.[2] This characteristic ensures that when the type of some component is given, any component implementation that conforms to the type can be used in the actual system without violating basic rules of component composition.

More abstract languages focus on modeling, and often their type systems have a different structure. For example, both the Acme ADL and the PVS theorem proving language [24] use type systems where types are arbitrary logical predicates. In these systems, one type is a subtype of another if the predicate of the first type implies the predicate of the second type.

Predicate-based type systems such as those in Acme and PVS are ideal for design-level modeling, because they allow architects to combine specifications in flexible and rich ways. For example, consider an abstract architecture that is a hybrid of the pipe-and-filter and repository architectural styles [11]. In this example, a filter component type has at least one input and one output port, while a client component in the repository style has at least one port to communicate with the repository. A component in this architecture might inherit specification information from both the filter and the repository client specifications, yielding a component that has at least three ports: two for communicating with other filters and one for communicating with the repository.

Unfortunately, examples like this cannot be expressed in the more limited, implementation-level type systems of concrete ADLs. A specification that a component has a port implies a requirement that the environment will match that port up with some other component, and therefore conventional type systems require a component type to list all of the ports it might possibly have (or at least all those ports that are expected to be connected at run time). There is no way to say that a filter component has "at least two ports"—instead, one must say that the filter has "at most" or "exactly" two ports. Therefore, one cannot combine the filter type with a repository component type (which defines a third port that is prohibited by the concrete filter specification) at the concrete level as one can in the abstract level.

[2]This distinction in the way required and provided services are handled is known as the contravariant subtyping principle.

Because of these differences in expressiveness, the subtyping relationships that hold at the abstract level may not be legal at the concrete level of architecture, requiring different type structures at the two levels of abstraction. It is also possible that a subtyping relationship that is not modeled at the abstract level could be added at the implementation level, simply because inheriting the functionality of one component is convenient when implementing another component.

2.4. Challenge: Complex Connectors

A distinguishing characteristic of architecture description languages is that they model architectures using explicit connectors, rather than bindings that implicitly connect one component to another [17]. Many abstract architecture description languages have rich connector modeling features, such as the ability to declare new connector types or define the substructure of connectors [10][16]. Other ADLs provide sophisticated models of connector behavior through process algebras [4].

Unfortunately, more concrete architecture description languages typically have weaker facilities for modeling connectors. For example, ArchJava allows developers to specify connector types and implementations, but neither supports an architecture-level decomposition of those connectors into more primitive pieces [7]. The UML 2.0 can express connector type information through UML association classes [13], but it does not support connector decomposition. Thus, when refining an abstract architectural view into a more concrete one, a key question is how to represent the abstract view's rich connector information in the context of the more concrete view.

2.5. Challenge: Overlapping Information

Abstract and concrete architectural views may have partially overlapping information, making it potentially more complicated to relate the two in consistent ways. For example, non-functional properties and style information in an abstract architecture may have no implementation artifact that can be directly associated with them. Abstract ADLs may support more than one representation for a component, but in a concrete, implementation-oriented view only one representation will be chosen. Similarly, a concrete architectural view may have interface and implementation details that are omitted in the abstract view.

Overlapping information is not necessarily a problem for translating an abstract view into a concrete one: the architect need only omit those parts of the abstract architecture that are not directly realized at the concrete level, and add any necessary interface and implementation information. However, building tools that keep these views consistent as both evolve over time can be quite challenging due to information overlap. In order to synchronize

abstract and concrete views that may have been individually modified, a tool must keep track of which information overlaps and which is common to the two views, so as to avoid losing information during the synchronization operation.

3. Refinement Strategies

In this section, we outline general strategies for addressing each of the challenges outlined in Section 2. In some cases, the refinement strategy is straightforward as there is a one-to-one correspondence between elements in the abstract and concrete views. However, there are a number of cases where refinement decisions need to be made by the architect. In many refinement tools, these decisions are made before the tools are written, providing the architect with only the option chosen by the tool designer. In general, however, different refinement decisions are appropriate in different circumstances, and so ideally a refinement tool should allow the architect to specify these decisions.

3.1. Type and Instance Mappings

The first refinement challenge is mapping relationships between types and instances from the abstract component and connector view to a more concrete, implementation-oriented view. Typically, component types and instances in the abstract view will have corresponding types and instances in the implementation view. However, as discussed earlier, there are a number of exceptions that the architect must be able to address.

For each element in the abstract view, the architect should be able to specify whether there is a corresponding element in the concrete view. For example, the Filter type in the Capitalize example is probably too generic to have any implementation code associated with it, because it has neither a concrete interface nor behavior of its own.

On the other hand, sometimes component instances in the abstract architecture correspond to both a type and an instance at the more concrete level. For example, the upper and lower filters in the Capitalize example have different behavior, and so each will probably have a corresponding type in the concrete architectural model. Thus, architects must also be able to specify whether a component instance should be translated into just a component instance or into a type/instance pair.

As described earlier, the type systems used in abstract and concrete architectural views may differ, so that refinement will not always preserve subtyping relationships between the types of architectural elements. For example, consider a small variation of the Capitalize architecture above, where the architect modeled Upper, Lower, Split, and Merge component types that subtype the Filter type. At the concrete level, it might be possible for Upper and Lower to subtype from Filter, but Split and Merge would be prohibited from subtyping from Filter because of their additional ports. This example illustrates that architects must be able to specify any changes in the inheritance hierarchy as part of the refinement strategy.

3.2. Connector Mappings

Typically, concrete models have relatively sophisticated facilities for representing components and ports, but weaker support for modeling connectors and roles. For example, the concrete C&C views provided by UML 2.0, and ArchJava do not allow developers to define the substructure of a connector. Thus one of the biggest refinement challenges is deciding how to refine connectors.

In simple cases, the built-in connectors provided by modeling languages may be sufficient. For example, both ArchJava and UML 2.0 provide default connector semantics that directly bind the required port of one component to the provided port of another. For connectors that have richer behavior, at least two general strategies are possible:

1. Implement a complex connector as a component in the concrete view, connected by simple connectors. This allows architects to leverage the concrete languages' considerable support for components, at the cost of blurring the distinction between components and connectors at the concrete level.
2. Leverage view-specific support for connectors, despite its limitations.

Only the first strategy can be used to model the substructure of a connector in languages that do not directly support rich connector representations. However, strategy two can have both design-level and implementation-level advantages. At the design level, it better preserves the architect's intent by distinguishing more clearly between components and connectors. It may also be beneficial at the implementation level due to implementation support for reuse of connector code. For example, object request brokers (ORB) allow developers to define customized, reusable connectors using hooks within the ORB infrastructure. Similarly, ArchJava, UML 2.0, and xADL 2.0 allow developers to specify a class that implements connector functionality.

3.3. Handling Information Overlap

Dealing with the challenges of overlapping or missing information in architectural refinement depends largely on the details of the abstract and concrete views in question. General strategies for handling information overlap include choosing information to omit, adding new architectural information as part of the refinement, and in some

cases, changing the form of architectural information to suit the new view.

For example, if a component in an abstract view has multiple representations, it is likely that only one of these representations will be chosen for the concrete view, with the others omitted. At the same time, the concrete view may add additional interface and implementation details that are absent from the abstract view. Finally, information on architectural style might be preserved by putting components from each relevant style into a corresponding implementation package or module. The refinement tool we have built, discussed in the next section, uses all of these refinement strategies.

4. Refining Acme to ArchJava

As discussed in Section 3, mapping between an abstract C&C view and a concrete C&C view involves choices about how to map specific elements. Because different choices may be appropriate in different circumstances, architects should be able to choose particular refinement strategies depending on the situation. This choice could be made by the architect as the architecture is being developed, or as part of the refinement process.

In our work, we chose to allow the architect to indicate the choices in the architectural model, rather than within the refinement tool. We believe that this has the following benefits:

1. The architect does not need to learn another tool to refine the architecture, and the choices are explicitly stored in the architectural model.
2. The architect can specify the choices incrementally, as the architecture is being developed.
3. The architectural tool can be used to check whether the architecture has sufficient detail to be refined, and not begin the refinement process until the architecture is ready for refinement.

We developed a prototype tool for refining abstract architectural descriptions in the Acme language into concrete architectural descriptions, represented as skeleton code in the ArchJava language. Programmers can then implement the actual behavior of the system within the ArchJava skeleton.

We begin this section by providing a brief introduction to Acme and ArchJava, explaining why we chose these languages as the source and target of refinement, and continue by describing the design of the refinement tool.

4.1. Abstract Architecture Description in Acme

Acme represents an abstract architectural model as an annotated, hierarchical graph. Nodes in the graph are *components*, which represent the principal computational elements and data stores of the system. Arcs are *connectors*, which represent the pathways of interaction between the components. Components and connectors have explicit interfaces (termed *ports* and *roles,* respectively). A system (or configuration) is defined as a set of components and connectors, in addition to attachments of ports to roles. To support various levels of abstraction and encapsulation, components and connectors can be hierarchically decomposed into *representations*.

To account for semantic properties of the architecture, elements in a system can be annotated with extensible property lists. Properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). Properties associated with a component might define its core functionality, performance attributes (e.g., average time to process a request, load, etc.), or its reliability.

In addition to representing generic systems, Acme allows architectural styles (or *families*) to be defined. An architectural style defines a set of types for components, connectors, roles, ports, and properties together with a set of rules that govern how elements of those types may be composed. An Acme system can declare itself to be in particular styles, which means that the elements in the system may use types defined by that style, and that the system satisfies the rules of that style. For example, the architecture in Figure 2 is in a Pipe-Filter architectural style. The component types available in this style are Filters, DataSinks, and DataSources; the only connector type is a Pipe. Port types discriminate between the writing and reading ends of a filters and data nodes, and role types between the source and sink ends pipes. Rules defined in the family say, for example, that pipes may only have two ends.

Acme's type system supports multiple inheritance, allowing an element to simultaneously extend types taken from multiple architectural styles. Acme's type system is predicate based, as discussed in section 2.3, so that an element is a subtype of any type whose properties and rules it satisfies. Architects can exploit Acme's predicate type system to specify refinement strategies, as described below.

Acme is an ideal source language for our tool because it embodies many of the characteristics of abstract design languages, including a predicate-based type system, extensible property support, rich modeling of connectors, and support for multiple representations of architectural elements. However, the principles behind our tool design apply to any abstract ADL with similar characteristics.

4.2. Concrete Architectures in ArchJava

ArchJava is an extension to the Java programming language that allows software engineers to specify a concrete software architecture within implementation code [2]. The research contribution of ArchJava is a novel type

```
family ArchJavaFam = {
  property isPreserved : boolean;

  property type GenerationPolicyT =
    enum {instanceOnly, typeOnly, instanceAndType};
  property type MethodSignaturesT = set{string};

  component type ArchJavaComponentT = {
    property extendParent : boolean;
    property classGenerationPolicy : GenerationPolicyT;
  }

  connector type ArchJavaConnectorT = {
    property isImplicit : boolean;
    property extendParent : boolean;
    property classGenerationPolicy : GenerationPolicyT;
  }

  port type ArchJavaPortT = {
    property provides : MethodSignaturesT;
    property requires : MethodSignaturesT;
  }
}
```

Figure 3. The ArchJava Architectural Style.

system, which statically ensures that the implementation of a software system conforms to the declared architecture.

ArchJava is representative of other concrete architectural views in the architectural features it supports, including components, ports and explicitly typed connectors. ArchJava components are run-time entities, not code modules, and the language supports rich forms of architectural dynamism. ArchJava's guarantee of architectural conformance makes it a particularly appropriate target for our refinement tool, because the tool can build on ArchJava to provide a conformance guarantee between an abstract Acme architecture and the Java code that implements the ArchJava architecture. However, the refinement strategies used in our tool are appropriate for many other concrete ADLs as well.

4.3. The ArchJava Refinement Style

When refining an abstract Acme architecture into a more concrete ArchJava architecture, the architect must make a number of refinement choices and express these to the refinement tool. We have chosen to leverage Acme's style support, using an ArchJavaFam style to define how such choices are expressed in Acme. Because Acme systems can satisfy multiple styles, we can "mix in" the ArchJavaFam style so that it can be used in conjunction with other styles. For example, if we wish to refine the model represented by Figure 2, it would need to satisfy both the Pipe-Filter family and the ArchJavaFam family (and filters would need to satisfy both the ArchJavaComponentT type and the FilterT type, for example).

Figure 3 shows the ArchJavaFam style, which defines how the architect expresses refinement decisions. All components that are intended to be refined to an ArchJava component must implement the ArchJavaComponentT type. This type defines two properties that allow the architect to express how the type hierarchy is mapped in the refinement process (sections 2.3 and 3.1):

- The classGenerationPolicy property determines whether an ArchJava class is generated corresponding to the Acme type (typeOnly), or if a class is generated corresponding to an Acme instance (instanceOnly), or if both classes are generated, with the instance class extending the type class (instanceAndType). This property allows ArchJava components to be generated for Acme instances (with or without extending the Acme type), in order to capture the behavior of different instances more effectively.
- If the extendParent property is true for some Acme component type A, and Acme component type A extends some other Acme component type B, the generated ArchJava type for A will extend the generated ArchJava type for B; otherwise, A will not extend B. This property allows the ArchJava inheritance hierarchy to differ from the Acme inheritance hierarchy. Note that the extendParent property is only applicable when the classGenerationPolicy property is either typeOnly or instanceAndType. When the classGenerationPolicy property is instanceOnly, no information from Acme's inheritance will be used.

Connectors have the same two properties as components, allowing the type hierarchies representing connectors to differ in the two languages. In addition to custom connector types [3], ArchJava provides a built-in connector (which binds required methods directly to provided methods) that can be selected by setting the isImplicit property to true.

To specify the details of a port's interface in Acme, an architect can specify a set of provided and required methods as properties of the port. When the corresponding port is generated in ArchJava, method stubs are added corresponding to these properties.

In addition to having to satisfy the structural characteristics of the family, there are rules defined that each system must satisfy. If any of these rules are unsatisfied, then refinement will not be allowed. Among these rules are:

1. If an element has more than one representation, then only one of them may have the isPreserved property set to true. This is the representation that will be mapped to an ArchJava implementation.

```
system capitalize : PipeFilterFam, ArchJavaFam = {

component lower : FilterT, ArchJavaComponentT = {
   property classGenerationPolicy = instanceAndType;
   property extendParent = false;
   port input : InputT, ArchJavaPortT = {
      property requires = {};
      property provides = {"void processChar (int c)"};
   }
   port output : OutputT, ArchJavaPortT = {
      property requires = {"void processChar (int c)"};
      property provides = {};
   }
}

connector pipe1 : PipeT, ArchJavaConnectorT = {
   property isImplicit = true;
   property classGenerationPolicy = typeOnly;
   property extendParent : false;
   roles {src, snk};
}

attach lower.output to pipe1.src;
attach merge.input1 to pipe1.snk
...
}
```

Figure 4. An Acme system to be refined.

2. All components, connectors, and ports must satisfy their corresponding ArchJavaFam types.
3. No connectors can be dangling.
4. When two ports are connected, their provides and requires method signatures must match.

Using the ArchJavaFam family allows the architect to specify decisions that need to be made in order for the Acme architecture to be refined into an ArchJava program. Our family definition provides defaults for each of these decisions, allowing an architect to omit properties that match the default rules. For example, in our default refinement strategy, components and connectors do not extend their parents in Acme, instance classes are generated for components and type classes are generated for connectors, and the set of requires and provides methods is empty. Our preliminary experience with the refinement tool suggests that these defaults can eliminate many of the property declarations that would otherwise be required.

4.4. Acme to ArchJava Refinement

Figure 4 shows how an architect could specify a portion of the architecture from Figure 2. For the lower component, the architect specifies a classGenerationPolicy of both,

```
public component class Filter = { … }

public component class Lower extends Filter {
   public port input {
      provides void processChar (int c) {}
   }
   public port output {
      requires void processChar (int c) {}
   }
}

public component class Capitalize extends Filter {
   private final Lower lower = new Lower ();

   connect lower.output, merge.input1;
   ...
}
```

Figure 5. The ArchJava generated from the Acme system in Figure 4.

meaning that the refinement will create the ArchJava component classes Filter and Lower (with Lower extending Filter), and the lower instance will be of type Lower. If the classGenerationPolicy was typeOnly then no Lower component class would be generated in ArchJava, and lower would be an instance of Filter. If the classGenerationPolicy was instanceOnly then the Lower component class would be generated, but would not extend Filter at the ArchJava level. In addition to this, the architect creates a Pipe (called pipe1), specifies that the built-in ArchJava connector will be used (by setting isImplicit to true), and then attaches one of the roles to the output port of lower.

The ArchJava skeleton resulting from the refinement is shown in Figure 5. A component class called Filter is generated, in addition to the Lower component class. The Lower component class has input and output ports with the appropriate required and provided methods. Another component class, Capitalize, is created for the system encompassing the architecture, and a field pointing to the subcomponent lower (of type Lower) is generated.

Finally, lower will be connected to the merge filter with a built-in connector using the **connect** command. Because of our use of ArchJava's built-in connectors, the concrete architecture does not explicitly name the pipe1 connector.

4.5. Implementation

We have implemented an Eclipse-based tool that allows architects to perform architectural refinement in the manner outlined in this paper. Our tool builds on Eclipse plugins that support development in Acme and ArchJava, respectively. The first plugin, AcmeStudio, provides graphical support for the development of architectural

styles and models in Acme [25]. The second plugin provides an integrated development environment for the ArchJava language.

The Acme-ArchJava refinement tool is also implemented as a plug-in to the Eclipse IDE framework, and acts as a bridge between AcmeStudio and ArchJava. Architects develop architectures in AcmeStudio according to the ArchJava style described in Section 4 (in addition to the architectural style they would normally use), and define the properties required by the ArchJava style. Once this is complete, and the rules of the ArchJava style are satisfied, an action in AcmeStudio is enabled that generates the required ArchJava skeleton code. This code can then be developed in the ArchJava Eclipse plugin.

We continue to refine the implementation of the tool to make it more robust, and plan to release it as open source software in November, 2004.

5. Related Work

Theory of Refinement. The theoretical basis of architectural refinement was discussed in the context of the SADL architectural description language. SADL formalizes architectures in terms of theories, providing a framework for refining an abstract architecture into a concrete one, while ensuring that the resulting architecture is consistent with the original architecture [19]. However, SADL provides no tool support for refining abstract architectures to more concrete ones, requiring architects to make this transition by hand.

More recent work on the theory of refinement includes Egyad et al.'s methodology for easing refinement by integrating refinement information into family architectures [8], and Baresi et al.'s modeling of architectural refinement in terms of graph rewriting rules [5].

Architecture to UML. Perhaps the most related work comes from the work mapping architectures to UML [1][9][15]. In this work, many of the issues in mapping connectors to a more concrete view are discussed but are limited to representing connectors as classes at the concrete level. They acknowledge the issue of mapping instances and types, but do not discuss this in detail.

Refinement Process. The OMG's Model-Driven Architecture provides a method and set of notations for moving between platform-independent and platform-dependent designs [21]. The MDA proscribes a two-level development process, in which deployment details are added at the low level so that the same abstract design can be used in different concrete settings. Work on the MDA compliments our work by investigating platform dependence issues that appear at a lower level of architectural abstraction.

Concrete Architecture to Code. A number of projects have looked at refining a concrete architecture to code, using code generation, library support, or integration with an implementation language. In the category of code generation, for example, UniCon provides tools that use an architectural specification to generate connector code that links components together [26].

In the category of library support, C2 and its successor, xADL 2.0, provides runtime libraries in C++ and Java that connect components together as specified in an architectural description [7][16]. Darwin also provides infrastructure support for implementing distributed systems specified in the Darwin ADL [14].

Furthermore, the Rapide system allows an architectural specification to be filled in with implementation code in an executable sub-language or in languages such as C++ or Ada [12]. The ArchJava language uses a similar strategy but builds an architecture description language into the Java implementation language rather than the other way around.

All of these code generation, library support, or implementation language integration systems assume that the starting point is a fairly concrete architectural view. Our work is complimentary in that it starts with an abstract architectural view and provides tool support for refining it into a more concrete view.

6. Conclusion and Future Work

This paper describes a number of key challenges in refining an abstract component and connector architectural view to a more concrete, implementation-oriented component and connector architectural view. We propose general strategies for addressing each of these challenges, and report on the design of a concrete tool that supports refinement from abstract architectures expressed in Acme to more concrete architectures expressed in ArchJava. We believe that this tool-supported refinement technology will be essential to the long-term vision of keeping an architectural design consistent with implementation as a program evolves.

In future work, we plan to enhance the refinement tool to support round-tripping between abstract Acme and concrete ArchJava representations of a software architecture. A key challenge will be synchronizing changes made separately to the two representations without losing the information that is particular to either representation.

References

[1] Marwan Abi-Antoun and Nenad Medvidovic. Enabling Refinement of a Software Architecture into a Design. UML '99.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to

Implementation. Proc. International Conference on Software Engineering, Orlando, Florida, May 2002.

[3] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. Proc. European Conference on Object-Oriented Programming, Darmstadt, Germany, July 2003.

[4] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, 6(3), July 1997.

[5] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. Style-Based Refinement of Dynamic Software Architectures. Proc. WICSA, 2004.

[6] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford. Documenting Software Architectures: Views and Beyond. SEI Series in Software Engineering, Addison Wesley, 2003

[7] Eric M. Dashofy, André van der Hoek, Richard N. Taylor. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. Proc. International Conference on Software Engineering, Orlando, Florida, May 2002.

[8] Alexander Egyed, Nikunj Mehta, and Nenad Medvidovic. Software Connectors and Refinement in Family Architectures. Proc. 3rd International Workshop on the Development and Evolution of Software Archtiectures for Product Families (IWSAPF), Spain, 2000.

[9] David Garlan, Andrew Kompanek, and Shang-Wen Cheng. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computing*, Elsevier Press, 2002.

[10] David Garlan, Robert Monroe, and Dave Wile. "Acme: Architectural Description of Component-Based Systems." Foundations of Component-Based Systems, Leavens, G.T, and Sitaraman, M. (eds), Cambridge University Press, 2000.

[11] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering, I (Ambriola V, Tortora G, Eds.) World Scientific Publishing Company, 1993.

[12] David C. Luckham and James Vera. An Event Based Architecture Definition Language. IEEE Trans. Software Engineering 21(9), September 1995.

[13] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmenl, and Jaime Oviedo Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, 2004.

[14] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. Proc. 5th European Software Engineering Conference (ESEC'95), Sitges, September 1995. (LNCS 989.)

[15] Nenad Medvidovic, David Rosenblum, David Redmiles, and Jason Robbins. Modeling Software Architectures in the Unified Modeling Language. TOSEM 11(1), 2002.

[16] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. Proc. Foundations of Software Engineering, San Francisco, CA, October 1996.

[17] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Software Engineering, 26(1), January 2000.

[18] Robert Monroe. "Capturing Architecture Design Expertise with Armani." Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163, 1998.

[19] Mark Moriconi, Xiaolei Qian, and Robert A. Riemenschneider. Correct Architecture Refinement. IEEE Trans. Software Engineering, 21(4), April 1995.

[20] Object Technology International Inc. "Eclipse Platform Technical Overview." http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

[21] Object Management Group. MDA: The Architecture of Choice for a Changing World. http://www.omg.org/mda.

[22] Object Management Group. UML 2.0 Superstructure Specification: Final Adopted Specification. OMG document ptc/08-03-02, 2003.

[23] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17:40--52, October 1992.

[24] John Rushby, Sam Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. IEEE Trans. Software Engineering 24(9), September 1998.

[25] Bradley Schmerl and David Garlan. Supporting Style-Centered Architecture Development. Proc. 26th International Conference on Software Engineering, Edinburgh, Scotland, 2004.

[26] Mary Shaw, Rob DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. IEEE Trans. Software Engineering, 21(4), April 1995.