Mapping Architectural Concepts to UML-RT

Shang-Wen Cheng <zensoul@cs.cmu.edu> *
(412) 268–3041, (412) 268–5576 (Fax)
David Garlan <garlan@cs.cmu.edu>
(412) 268–5056, (412) 268–5576 (Fax)
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3890

Abstract

Complex software systems require expressive notations for representing their software architectures. Two competing paths have emerged, one using a specialized notation for architecture—or architecture description language (ADL), the other using notations applied generally throughout design, such as UML. The latter has a number of benefits, including familiarity to developers, close mappings to implementations, and commercial tool support. However, it remains an open question how best to use object-oriented notations for architectural description and whether they are sufficiently expressive as currently defined. In this paper, we present a mapping between Acme—a notation designed for expressing architectures—and the UML Real-Time Profile—an object-oriented design notation. Specifically, we describe (a) how to map Acme descriptions to descriptions in the UML Real-Time Profile, and (b) the places where this mapping breaks down.

Keywords
Software architecture, mapping, Acme, UML-RT

1 Introduction

Software architecture has emerged as a critical level of abstraction in the description of a complex system [14]. At the architectural level, one models the principal system components and their pathways of interaction. An architectural description provides a high-level, hierarchical view of a system and permits design-time reasoning about system-level properties, such as performance, reliability, portability, and conformance to external standards and architectural styles.

Today, most architectural descriptions are described informally using box-and-line diagrams with explanatory prose and idiosyncratic, usually project-specific, visual conventions. Consequently, architectural descriptions are poorly understood by developers; they cannot be analyzed for consistency or completeness; they are only hypothetically related to implementations; their properties cannot be enforced as a system evolves; and they cannot be supported by tools to help software architects with their tasks.

A number of researchers have suggested improving today's practice through the use of more standardized and formal notations for architectural description. There are two main sources of such recommendation: one from the software architecture research community, and the other from the object-oriented community. The first has proposed a number of "architecture description languages" (ADLs) specifically designed to represent software and system architectures [8]. These languages have matured over the past six years. Most come with tool sets that support many aspects of architectural design and analysis .

The object-oriented community, on the other hand, has proposed to use general-purpose object modeling notations, such as UML. Indeed, the creators of UML specifically identify architectural modeling as one of their primary capabilities. However, despite those claims, there has been considerable debate over whether object-oriented modeling notations are appropriate vehicles for architectural descriptions, and if so, how best to use them for that purpose. A number of papers have made specific proposals, by indicating which standard UML model elements should be used to model which architectural concepts [5, 7, 6]. Unfortunately, each such paper suggested a different approach, often reflecting the auther's specific needs for modeling. To understand the design space of mappings, Garlan in [3] attempted to produce a taxonomy of such approaches and outline strengths and weaknesses of each. The conclusion was that (a) there is no single "best" way; and (b) none of the approaches is ideal.

In this paper we adopt a somewhat different approach: rather than using *generic* UML, we start by leveraging the work done in defining a specific UML *Profile* [1]—namely the "Real-Time" Profile, UML-RT. The benefit of doing this, is that the authors of that profile have already given considerable thought to modeling the runtime structures and behaviors of complex systems. In particular, they adopt the notion of a connector between components as a protocol, a point of view closely aligned with current thinking in the ADL research community. We argue that this profile, therefore, provides a particu-

*Author will present paper if accepted

[1] A UML Profile is a "standardized" language extension, typically created for a particular class of system or domain.

larly natural home for architectural modeling.

In the remainder of this paper, we develop this argument by showing how the foundational concepts of architecture are naturally represented in UML-RT. We start by explicitly enumerating the basic ingredients of architectural models, as found in modern ADLs. These are the features of a standard architecture interchange language, called Acme [4], introduced in the next section. Next we outline the key features of UML-RT. Then we show show the former can be mapped to the latter. We then consider places where the mapping is incomplete, or semantically unsatisfying. Finally, we use those observations to conclude by outlining prospects for improved capabilities in UML that will support architectural modeling even better.

2 Architectural Description and Acme

The software architecture of a system models its high-level structure as a collection of interacting components. Currently, there is considerable diversity in the ways that practitioners represent architectures, but most depend on box-and-line diagrams. Thus, a number of ADLs have been proposed over the past decade to put architectural description on a more solid notational and semantic footing [3]. While these languages differ in details, capabilities, and tool support, a general consensus about the main ingredients of architectural description has emerged. Among the shared "design elements" are seven core concepts: components, connectors, systems, properties, styles, representations, and rep-maps. We briefly describe each below:

- *Components* represent the primary computational elements and data stores of a system and correspond intuitively to the boxes in the informal box-and-line diagrams. Typical examples of components include filters, clients, servers, blackboards, databases. Components may have multiple interfaces (called *ports*) that define points of interaction between a component and its environment. A component may have several ports of the same type (e.g., 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—a pipe, procedure call, and event broadcast—and complex interactions, such as a client-server protocol or a SQL link between an application and a database. A connector has interfaces (called *roles*) that define the roles played by the participants in the interaction.

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

- *Properties* represent additional information beyond structure about the parts of an architectural description. They are typically used to represent anticipated or required extra-functional aspects of an architectural design, and it is generally desirable to be able to associate properties with any architectural element in a description (components, connectors, systems, and their interfaces). For instance, a property associated with an interface (port or role) may describe its interaction protocol.

- *Types* are predicates that define a collection of related design elements. A *Style* represents a family of related systems, and defines a collection (design vocabulary) of *types*—as a set of component, connector, port, role, binding (see *Rep-map*), and property types—together with rules for composing instances of the types [14]. Examples include data-flow architectures based on graphs of pipes and filters, blackboard architectures, and layered systems.

- *Representations* are more detailed, lower-level descriptions of components or connectors. One or more such descriptions may be defined for each component or connector to support hierarchical descriptions of architectures. The ability to associate multiple representations with a design element allows Acme to encode multiple views of architectural entities .

- A *Rep-map* (short for "representation map") defines the correspondence between the internal system representation and the external interfaces of the component or connector that is being represented. In the simplest case, a rep-map provides an association between internal ports and external ports (or, for connectors, internal roles, and external roles). Such associations are termed *bindings*. Considerably more complex maps are possible for other cases.

Acme is a second-generation ADL that builds on the experience of other ADLs and embodies the architectural ontology described above, providing a semantically extensible language and a rich tool-set for architectural analysis and integration of independently developed tools. Acme supports the definition of four distinct aspects of architecture: (1) Structure—the organization of a system into its constituent parts; (2) Properties of interest—information about a system or its parts that allow one to reason abstractly about overall behavior; (3) Constraints—guidelines for how the architecture can

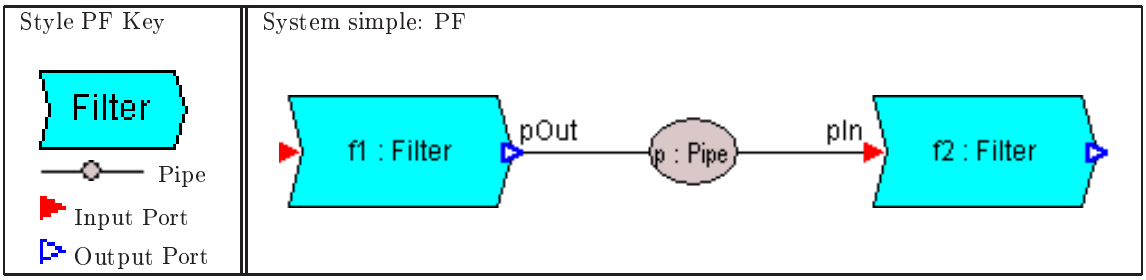


Figure 1: A simple system in the Pipes & Filters style (Generated in AcmeStudio v.1.5b)

change over the life-cycle of the software; (4) Types and styles—defining families of architecture.

To illustrate the use of these architectural concepts, consider the simple example shown in Figure 1, which defines a simple string-processing application consisting of two processing steps. This system is described in a pipe-and-filter style, which provides a design vocabulary, shown in the textual description in Figure 2, consisting of a `Filter` component type, `Pipe` connector type, and input and output `Port` types. In addition, a constraint ensures that each connector defined in a system using this style is of type `Pipe`. Properties of the components list performance characteristics used by a tool to calculate overall system throughput.

```
Family PipesAndFiltersFam = {
    Port Type InPort;
    Port Type OutPort;
    Role Type Source;
    Role Type Sink;
    Component Type Filter = {
        Port in : InPort;
        Port out : OutPort;
        Property throughput : int;
    };
    Connector Type Pipe = {
        Role src : Source;
        Role snk : Sink;
    };
    Property Type StringMsgFormat =
        Record [ size:int; msg:String;];
    Invariant Forall c in self.Connectors
        @ HasType(c, Pipe);
};
System simplePF : PipesAndFiltersFam = {
    Component f1 : Filter = new Filter;
    Component f2 : Filter = new Filter;
    Connector p : Pipe = new Pipe;
    Attachments {
        f1.out to p.src;
        f2.in to p.snk;
    };
}; /* end system */
```

Figure 2: Textual description of Simple PF showing the family and the system definition.

3 UML-RT and UML Background

UML unifies a number of object modeling notations in a common framework for visualizing, specifying, constructing, and documenting object-oriented systems. It is quickly becoming a standard object notation for object-oriented design [1, 10]. Some of the principal constructs of UML, known as model *elements*, include classes, interfaces, objects, components, packages, relationships, and stereotypes [9]. These constructs can be composed in various ways in UML models and visualized in diagrams. Textual *annotations* may be associated with any of them, and these frequently appear in the form of arbitrary attribute-value pairs called *tagged values*. UML also defines a set of models for describing the dynamic behavior of a system, including *use cases*, *state machines* descriptions, and *collaboration diagrams* that specify system behavior using event-based *interaction scenarios*. Due to space constraints, we will assume some familiarity with UML.

The UML Real-Time Profile (UML-RT) addresses modeling concepts that have proven suitable for modeling the run-time architectures of complex real-time systems in application domains such as telecommunications, aerospace, and industrial control [12]. Specifically, UML-RT adds five stereotypes to standard UML to facilitate the modeling of run-time structures, which we briefly describe below.

- *Capsule* models a complex active object that may have multiple interface points (ports) through which it interacts with its environment. The capsule stereotype maps to a UML Class with the constraints that (1) it is always an active object; (2) it can have at most one state machine; (3) it can only have protected operations; (4) it may have attributes that are ports, connectors, or sub-capsules, but all these attributes have protected visibility except ports, which have public visibility. Thus, a capsule can contain sub-capsules, and these are related by a composition relationship in the corresponding class diagram.

- *Connector* is a communication object that conveys messages between ports attached to its ends. The message exchange conforms to the protocol associated with the connector. The connector stereotype maps to a UML AssociationClass, with the additional constraints that (1) the connector must have

exactly one protocol, (2) the connector must have the same number of association ends as its protocol has protocol roles, (3) the association ends must be attached to ports, and (4) the association ends of a connector must be attached to ports of the matching protocol role.

- *Port* serves as an interaction point for capsules. Ports can both receive and be the source of signals and synchronous calls. A port is either an end port (attached to a state machine) or a relay port (connected to a port on a sub-capsule). A port stereotype maps to a UML Class which (1) realizes exactly one protocol role and (2) can only appear as an attribute of a capsule class.

- *Protocol* is a generic specification of a closed group of participants (protocol roles) that interact in specific ways to achieve one or more desired objectives. The protocol stereotype maps to a UML Collaboration.

- *Protocol role* specifies one party in a protocol specification. A protocol role stereotype maps to a UML ClassifierRole with the constraints that (1) it does not have any features or available contents and (2) the base classifier of a protocol role is itself.

Table 1 gives a summary of the stereotypes. Details of this profile can be found in [11].

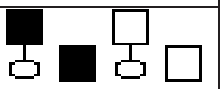
Stereotype	Base UML Class	Stereotype Icon
≪capsule≫	Class	
≪connector≫	AssociationClass	none defined
≪port≫	Class	
≪protocol≫	Collaboration	
≪protocolRole≫	ClassifierRole	

Table 1: Summary of the UML-RT stereotypes (The four stereotype icons for port, from left to right, are: *End port*, *relay or indeterminate port*, *conjugate end port*, and *conjugate relay or indeterminate port*.)

4 Mapping between Acme and UML-RT

Acme offers the language and tool support for creating formal architectural descriptions of desired systems and reasoning about the overall system properties. UML offers a variety of object modeling notations in a common framework with close mapping to implementation. It is familiar to most developers and has commercial tool support. Capturing the best of both methodologies helps to fill in the rather significant chasm between design at the highest level and implementation at the lower level. We aim to achieve this union by presenting a strategy for mapping between the notations of Acme and UML by way of UML-RT. While the mapping is still under development, the ultimate goal is to produce a tool for developers to transition smoothly between the respective tool environments.

4.1 A Mapping Strategy

We approach the mapping by considering both types and instances of seven basic elements of Acme: component, port, connector, role, system, representation, and rep-map. We present the rationales behind the mapping in this section, and discuss mismatches in the next section. Table 2 shows a summary of the mapping presented in this paper.

Acme components map naturally to UML-RT capsules since both represent primary computational elements, both have interfaces, and both can be hierarchically decomposed. Acme component types map to UML capsule-stereotyped classes, while component instances map to capsule-stereotyped objects (in a collaboration diagram).

Acme ports map to UML-RT ports since both serve as interfaces that define points of interaction between the computational elements and the environment. Acme port instances map to UML port-stereotyped objects. Acme port types could likewise be mapped to port-stereotyped implementation classes, but a UML-RT protocol role defines the *type* of the port [13], so we choose to map Acme port types to protocolRole-stereotyped classes in UML.

Acme connectors map to UML-RT connectors because both represent interactions between the computational units. Acme connector types map to the UML AssociationClasses, and Acme connector instances map to UML link (an instance of UML association). UML-RT protocols represent the behavioral aspects of UML-RT connectors, and the closest equivalent in Acme are constraints (and perhaps properties) applied to the Acme connectors themselves.

Although a UML-RT protocol is associated with a connector, the protocolRole serves as the Acme port type, so the more natural mapping of Acme roles to UML-RT protocolRoles causes a conflict. Since Link has LinkEnds in UML, and UML-RT connector instances map to UML links, we will map Acme role instances to UML LinkEnds and Acme role types to UML AssociationEnds.

Acme systems describe the structural configuration, as do UML-RT collaborations; thus, systems map to collaborations.

An Acme representation actually contains a system instance; thus, it can be mapped to a UML-RT collaboration.

An Acme binding associates an internal port of a component with its external port (and similarly for internal and external roles of a connector), while the UML-RT connector is used for the same purpose to link a sub-capsule port to a port of the enclosing capsule. Thus,

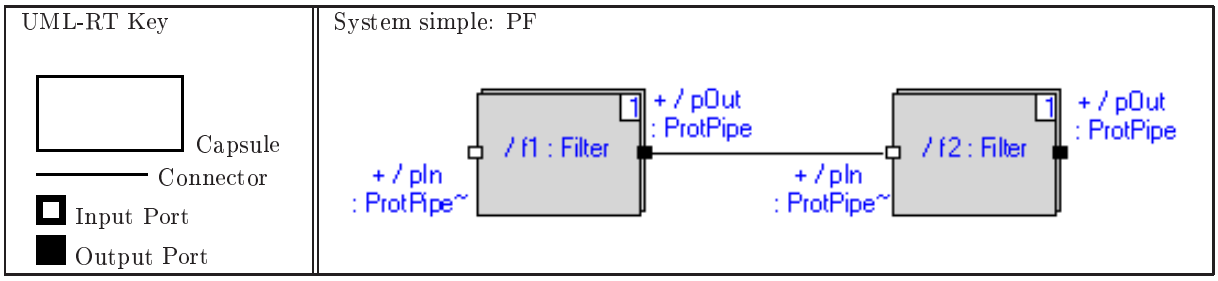


Figure 3: UML-RT collaboration diagram for simple PF system (Generated in Rational Rose Real-Time v. 2001)

Acme binding maps to UML-RT connector. However, there is no direct mapping for the Acme rep-map, and we will discuss this in Section 5.

4.2 An Example Translation to UML-RT

To illustrate this mapping, Figure 3 shows the simple pipe-filter system of Figure 1, but now drawn in UML-RT using the strategy we just outlined. A collaboration diagram and a class diagram are shown to represent the instance and the type relationships for the simple PF system, respectively.

In Figure 3, the filters *f1* and *f2* become capsules of type `Filter`, each with an input and an output port. A slash prepending the name denotes a role in a collaboration. Pipe *p* in the Acme diagram becomes a connector that conforms, in this case, to a pipe protocol (`ProtPipe`) with a `source` and a `sink` protocol role. The output and input Acme ports, joined by the connector, therefore play the `source` and `sink` protocol roles, respectively. Since a UML-RT port plays a specific role in some protocol, the protocol role defines the type of the port (which simply means that the port implements the behavior specified by that protocol role) [13]. Thus, *pOut*'s type is `ProtPipe::source`, and that of *pIn* is `ProtPipe::sink`.

For binary protocols, UML-RT provides notational conventions for the port icon and type name. The role selected as the base protocol role (in this case, the source role) is shown as a black-filled box, with the type denoted only by the protocol name, while the other, conjugate role, is shown as a white-filled box, with the type denoted by appending a '~' to the protocol name, as shown in the figure. UML-RT does not treat roles and ports separately; so, as discussed in the next section, we have no clean mapping for the Acme roles in UML-RT.

Since there is only one `Filter` type in the simple PF system, there is only one class in the class diagram shown in figure 4.2. In UML-RT, all elements contained by a capsule are considered attributes of that capsule class, and all attributes have protected visibility (indicated by a key and a shaded block next to the attribute) except ports, which have public visibility (indicated by just a shaded block). Additionally, ports are listed in a separately named compartment. The ≪capsule≫-stereotyped `Filter` class has two ports and a *throughput* value as attribute (obtained from the Acme text description). The connector in the collaboration diagram is abstracted as an association of the `Filter` class to itself, with roles *f1* and *f2* on either end of the association and 1 to 1 respective multiplicity.

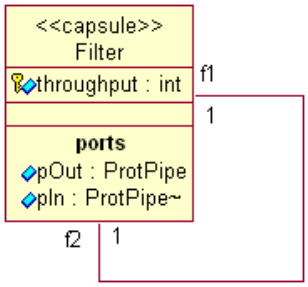


Figure 4: UML-RT class diagram for simple PF system

5 Mismatches

While the mapping presented in Section 4 provides a natural correspondence between most elements of Acme and UML-RT, the translation is not completely satisfactory. There are several semantic and syntactic mismatches to resolve, perhaps more than there are matches. We now discuss the mismatches and offer possible solutions to resolve some of them. Most of these issues will require further investigation.

5.1 Component Mismatches

Acme components seem the least troublesome to map to UML-RT. UML-RT capsules support the hierarchical decomposition of Acme components via class composition relationships, with the subtle difference that UML-RT treats the enclosing capsule as a class instead of an instance in a collaboration diagram. Essentially, UML-RT's collaboration diagram of decomposition conveys a *pattern*, such that any instance of the enclosing capsule will have the prescribed sub-capsules. This is a semantic mismatch because components in Acme systems are instances rather than patterns.

In UML-RT, certain sub-capsules act as *plug-ins* that are dynamically *imported*, along with the instantiation of

	Acme	UML-RT	Equivalent UML
1	**Component**	≪**Capsule**≫	≪capsule≫ object
	Type	≪**Capsule**≫ class	≪capsule≫ class
2	**Port**	≪**Port**≫	≪port≫ object
	Type	≪**ProtocolRole**≫	≪protocolRole≫ abstract class (≪port≫ implem. class)
3	**Connector**	≪**Connector**≫ (link)	Link (binary or n-ary)
	Type	Association class	AssociationClass
	(Behavioral constraint)	≪**Protocol**≫	≪protocol≫ class (Collaboration)
4	**Role**	No explicit mapping/***ProtocolRole?***	
		Implicit elements:	
		LinkEnd	LinkEnd
	Type	AssociationEnd	AssociationEnd
5	**System**	Collaboration	Collaboration
6	**Representation**	Collaboration	≪capsule≫ object
7	**Binding**	≪**Connector**≫	Link

Table 2: Summary of Mapping from Acme to UML-RT (ordered by instance, then type, if present)

attached connectors, at run time. Plug-ins are not necessarily owned by the enclosing capsule. They allow for the modeling of dynamically changing structures while ensuring that all valid communication and containment relationships between capsules are specified explicitly [13]. It is not clear how plug-ins can be mapped to elements in Acme, because Acme does not have similar capability. Another feature not supported by Acme is UML-RT's provision for specifying capsule multiplicity.

One final mismatch concerns how a capsule's state machine, connected to its end port(s), can be captured in Acme components, which do not have explicit behavior models. Plausible devices include adding the constraint to the Acme system as properties, or as Wright properties, which would allow Wright tools to run checks on the system.

5.2 Port Mismatches

UML-RT requires two tags for the port: *isRelay* and *isInternal*, which are not explicitly provided by Acme port, but could be specified as Acme properties. A relay port in UML-RT is equivalent to, and could be mapped to, an Acme port that bridges an external connector to its internal binding within the port's component. An end port in UML-RT could be mapped to an Acme port without internal binding. However, there is no direct mapping in Acme for the state machine connected to a UML-RT end port. For the *isInternal* tag, Acme ports that appear on subcomponents of a component X can be mapped as internal UML-RT ports of capsule X, while Acme ports of X can be mapped as external UML-RT ports.

Although UML-RT's base-conjugate port notation is not available in Acme, defining different visualization properties for different ports produces the same results. Port multiplicity (an indicator of instance count) is yet another UML-RT feature not supported by Acme. Finally, UML-RT's attachement of a protocol role to its port restricts the port as well as the role (see 5.3).

5.3 Connector and Role Mismatches

Acme treats connectors as first-class entities, while UML-RT does not, so important semantic elements will be lost when translated to UML-RT models. In UML-RT class diagrams, connectors are not distinguished from other associations, which can lead to ambiguities, and they cannot exist on their own. In addition, a UML-RT connector must have a protocol associated with it to coordinate the behavior of the connected capsules, where as Acme's connector already provides that coordination. Finally, the decomposition of UML-RT connectors—i.e., elaborating subclasses and the collaboration of their instances—though theoretically possible, makes a challenging task because both the connector and the protocol (and thus the collaboration) must be decomposed. By the same token, translation of an Acme connector's representation to the equivalent UML-RT models becomes more difficult and mismatch-prone.

Acme roles present another set of mismatches because they really have no equivalent in UML-RT. Their closest match appears to be UML-RT protocolRoles, which are already used for port types. This treatment in UML-RT effectively reduce roles and ports to the same element. In contrast, ports and roles in Acme need only be consistent, but are not required to be identical.

5.4 Representation and Rep-map Mismatches

A UML-RT collaboration diagram of a capsule's innards shows interactions of subcapsule instances (as collabo-

ration roles), whose composition relationship with the parent capsule has already been established (plug-ins notwithstanding). Consequently, a capsule, given its class hierarchy, can only have one collaboration. Acme components, in contrast, can have many representations, each of which could exhibit different numbers of subcomponents, composed in different styles. Thus, UML-RT collaborations do not provide the same capability and flexibility. On the other hand, as mentioned in component mismatches, decompositions of a UML-RT capsule essentially convey a pattern for the instantiation of the enclosing capsule (because the enclosing capsule is shown as a class). However, Acme representation is not a pattern, but an instance.

UML-RT does not have an equivalent notion of Acme rep-map; instead, it models the elements of a rep-map, the bindings, as connectors. Mapping bindings as UML-RT connectors adds extraneous semantics by giving bindings protocols and attributes.

6 Conclusion

In this paper, we have demonstrated the feasibility of translating architectural descriptions to the UML-RT Profile. We illustrated the mapping with a simple example of a pipe-and-filter system. On balance, the mapping works well; specifically, the architectural concepts of component, port, connector, system, and representation have respective homes in UML-RT. However, as we noted, there are several constructs in UML-RT not provided in Acme, and vice versa. Among the problems, several could be resolved by augmenting Acme, while others do not have simple resolutions, such as the ability to describe multiple representations, and the notion of bindings as separate from connectors. Furthermore, we have not addressed the mapping of Acme style in this discussion. These issues require further analysis, resolution, and perhaps compromise of needs.

Ultimately, it would simplify matters to have homes in generic UML for the architectural concepts. OMG is currently working to incorporate better support for architectural modeling into new versions of UML. Undoubtedly, attempting to add new elements with as little disruption as possible to the existing UML metamodel presents many challenges. However, the improved model would greatly ease the mapping between Acme and generic UML, and reduce the need for translation by way of a specific profile.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.

[2] P. Donohoe, editor. *Proceedings of the TC2 1st Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, USA, Feb. 22–24 1999. Kluwer Academic Publishers.

[3] D. Garlan and A. J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In A. Evans and S. J. H. Kent, editors, *Proceedings, UML 2000—The Unified Modeling Language: Advancing the Standard: Third International Conference*, York, UK, Oct. 2–6 2000. Springer.

[4] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural descriptions of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[5] C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with UML. In Donohoe [2], pages 145–159.

[6] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21-6, pages 24–32, San Francisco, CA, USA, Oct. 16–18 1996. ACM Press.

[7] N. Medvidovic and D. S. Rosenblum. Assessing the suitability of a standard design method for modeling software architecture. In Donohoe [2], pages 161–182.

[8] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Software engineering: ESEC/FSE '97: 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, September 22–25, 1997: proceedings*, New York, NY, USA, 1997. Springer-Verlag Inc.

[9] OMG. *OMG Unified Modeling Language Specification, v. 1.3*, June 1999.

[10] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, 1999.

[11] B. Selic. UML-RT: A profile for modeling complex real-time architectures. Draft, ObjecTime Limited, Dec. 1999.

[12] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[13] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. http://www.objectime.com/otl/technical/, Mar. 1998.

[14] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.